HPX Documentation

v1.10.0

The STE||AR Group

May 29, 2024

USER DOCUMENTATION

If you're new to *HPX* you can get started with the *Quick start* guide. Don't forget to read the *Terminology* section to learn about the most important concepts in *HPX*. The *Examples* give you a feel for how it is to write real *HPX* applications and the *Manual* contains detailed information about everything from building *HPX* to debugging it. There are links to blog posts and videos about *HPX* in *Additional material*.

You can find a comprehensive list of contact options on Support for deploying and using HPX[1]. Do not hesitate to contact us if you can't find what you are looking for in the documentation!

See *Citing HPX* for details on how to cite *HPX* in publications. See *HPX users* for a list of institutions and projects using *HPX*.

There are also available a PDF version of this documentation as well as a Single HTML Page.

[1] https://github.com/STEllAR-GROUP/hpx/blob/master/.github/SUPPORT.md

CHAPTER

ONE

WHAT IS *HPX*?

HPX is a C++ Standard Library for Concurrency and Parallelism. It implements all of the corresponding facilities as defined by the C++ Standard. Additionally, in *HPX* we implement functionalities proposed as part of the ongoing C++ standardization process. We also extend the C++ Standard APIs to the distributed case. *HPX* is developed by the STE||AR group (see *People*).

The goal of *HPX* is to create a high quality, freely available, open source implementation of a new programming model for conventional systems, such as classic Linux based Beowulf clusters or multi-socket highly parallel SMP nodes. At the same time, we want to have a very modular and well designed runtime system architecture which would allow us to port our implementation onto new computer system architectures. We want to use real-world applications to drive the development of the runtime system, coining out required functionalities and converging onto a stable API which will provide a smooth migration path for developers.

The API exposed by *HPX* is not only modeled after the interfaces defined by the C++11/14/17/20 ISO standard. It also adheres to the programming guidelines used by the Boost collection of C++ libraries. We aim to improve the scalability of today's applications and to expose new levels of parallelism which are necessary to take advantage of the exascale systems of the future.

CHAPTER

TWO

WHAT'S SO SPECIAL ABOUT *HPX*?

- HPX exposes a uniform, standards-oriented API for ease of programming parallel and distributed applications.
- It enables programmers to write fully asynchronous code using hundreds of millions of threads.
- HPX provides unified syntax and semantics for local and remote operations.
- HPX makes concurrency manageable with dataflow and future based synchronization.
- It implements a rich set of runtime services supporting a broad range of use cases.
- HPX exposes a uniform, flexible, and extendable performance counter framework which can enable runtime adaptivity
- It is designed to solve problems conventionally considered to be scaling-impaired.
- HPX has been designed and developed for systems of any scale, from hand-held devices to very large scale systems.
- It is the first fully functional implementation of the ParalleX execution model.
- HPX is published under a liberal open-source license and has an open, active, and thriving developer community.

2.1 Quick start

The following steps will help you get started with *HPX*. Before getting started, make sure you have all the necessary prerequisites, which are listed in _prerequisites. After *Installing HPX*, you can check how to run a simple example *Hello, World!*. *Writing task-based applications* explains how you can get started with *HPX*. You can refer to our *Migration guide* if you use other APIs for parallelism (like OpenMP, MPI or Intel Threading Building Blocks (TBB)) and you would like to convert your code to *HPX* code.

2.1.1 Installing *HPX*

The easiest way to install *HPX* on your system is by choosing one of the steps below:

#.* * vcpkg * *

You can download and install *HPX* using the vcpkg[2] dependency manager:

```
$ vcpkg install hpx
```

#.* * Spack * *

Another way to install *HPX* is using Spack[3]:

[2] https://github.com/Microsoft/vcpkg
[3] https://spack.readthedocs.io/en/latest/

```
$ spack install hpx
```

#.* * Fedora * *

Installation can be done with Fedora[4] as well:

```
$ dnf install hpx*
```

#.* * Arch Linux * *

HPX is available in the Arch User Repository (AUR)[5] as hpx too.

More information or alternatives regarding the installation can be found in the *Building HPX*, a detailed guide with thorough explanation of ways to build and use *HPX*.

2.1.2 Hello, World!

To get started with this minimal example you need to create a new project directory and a file CMakeLists.txt with the contents below in order to build an executable using CMake[6] and *HPX*:

```
cmake_minimum_required(VERSION 3.19)
project(my_hpx_project CXX)
find_package(HPX REQUIRED)
add_executable(my_hpx_program main.cpp)
target_link_libraries(my_hpx_program HPX::hpx HPX::wrap_main HPX::iostreams_component)
```

The next step is to create a main.cpp with the contents below:

```
// Including 'hpx/hpx_main.hpp' instead of the usual 'hpx/hpx_init.hpp' enables
// to use the plain C-main below as the direct main HPX entry point.
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

int main()
{
    // Say hello to the world!
    hpx::cout << "Hello World!\n" << std::flush;
    return 0;
}
```

Then, in your project directory run the following:

```
$ mkdir build && cd build
$ cmake -DHPX_DIR=</path/to/hpx/installation> ..
$ make all
$ ./my_hpx_program
```

```
$ ./my_hpx_program
Hello World!
```

The program looks almost like a regular C++ hello world with the exception of the two includes and hpx::cout.

[4] https://fedoraproject.org/wiki/DNF
[5] https://wiki.archlinux.org/title/Arch_User_Repository
[6] https://www.cmake.org

- When you include `hpx_main.hpp` *HPX* makes sure that `main` actually gets launched on the *HPX* runtime. So while it looks almost the same you can now use futures, `async`, parallel algorithms and more which make use of the *HPX* runtime with lightweight threads.
- `hpx::cout` is a replacement for `std::cout` to make sure printing never blocks a lightweight thread. You can read more about `hpx::cout` in *The HPX I/O-streams component*.

Note:

- You will most likely have more than one `main.cpp` file in your project. See the section on *Using HPX with CMake-based projects* for more details on how to use `add_hpx_executable`.
- `HPX::wrap_main` is required if you are implicitly using `main()` as the runtime entry point. See *Re-use the main() function as the main HPX entry point* for more information.
- `HPX::iostreams_component` is optional for a minimal project but lets us use the *HPX* equivalent of `std::cout`, i.e., the *HPX The HPX I/O-streams component* functionality in our application.
- You do not have to let *HPX* take over your main function like in the example. See *Starting the HPX runtime* for more details on how to initialize and run the *HPX* runtime.

Caution: Ensure that *HPX* is installed with `HPX_WITH_DISTRIBUTED_RUNTIME=ON` to prevent encountering an error indicating that the `HPX::iostreams_component` target is not found.

When including `hpx_main.hpp` the user-defined `main` gets renamed and the real `main` function is defined by *HPX*. This means that the user-defined `main` must include a return statement, unlike the real `main`. If you do not include the return statement, you may end up with confusing compile time errors mentioning `user_main` or even runtime errors.

2.1.3 Writing task-based applications

So far we haven't done anything that can't be done using the C++ standard library. In this section we will give a short overview of what you can do with *HPX* on a single node. The essence is to avoid global synchronization and break up your application into small, composable tasks whose dependencies control the flow of your application. Remember, however, that *HPX* allows you to write distributed applications similarly to how you would write applications for a single node (see *Why HPX?* and *Writing distributed applications*).

If you are already familiar with `async` and `future` from the C++ standard library, the same functionality is available in *HPX*.

The following terminology is essential when talking about task-based C++ programs:

- lightweight thread: Essential for good performance with task-based programs. Lightweight refers to smaller stacks and faster context switching compared to OS threads. Smaller overheads allow the program to be broken up into smaller tasks, which in turns helps the runtime fully utilize all processing units.
- `async`: The most basic way of launching tasks asynchronously. Returns a `future<T>`.
- `future<T>`: Represents a value of type `T` that will be ready in the future. The value can be retrieved with `get` (blocking) and one can check if the value is ready with `is_ready` (non-blocking).
- `shared_future<T>`: Same as `future<T>` but can be copied (similar to `std::unique_ptr` vs `std::shared_ptr`).
- continuation: A function that is to be run after a previous task has run (represented by a future). `then` is a method of `future<T>` that takes a function to run next. Used to build up dataflow DAGs (directed acyclic

graphs). shared_futures help you split up nodes in the DAG and functions like when_all help you join nodes in the DAG.

The following example is a collection of the most commonly used functionality in *HPX*:

```cpp
#include <hpx/algorithm.hpp>
#include <hpx/future.hpp>
#include <hpx/init.hpp>

#include <iostream>
#include <random>
#include <vector>

void final_task(hpx::future<hpx::tuple<hpx::future<double>, hpx::future<void>>>)
{
    std::cout << "in final_task" << std::endl;
}

int hpx_main()
{
    // A function can be launched asynchronously. The program will not block
    // here until the result is available.
    hpx::future<int> f = hpx::async([]() { return 42; });
    std::cout << "Just launched a task!" << std::endl;

    // Use get to retrieve the value from the future. This will block this task
    // until the future is ready, but the HPX runtime will schedule other tasks
    // if there are tasks available.
    std::cout << "f contains " << f.get() << std::endl;

    // Let's launch another task.
    hpx::future<double> g = hpx::async([]() { return 3.14; });

    // Tasks can be chained using the then method. The continuation takes the
    // future as an argument.
    hpx::future<double> result = g.then([](hpx::future<double>&& gg) {
        // This function will be called once g is ready. gg is g moved
        // into the continuation.
        return gg.get() * 42.0 * 42.0;
    });

    // You can check if a future is ready with the is_ready method.
    std::cout << "Result is ready? " << result.is_ready() << std::endl;

    // You can launch other work in the meantime. Let's sort a vector.
    std::vector<int> v(1000000);

    // We fill the vector synchronously and sequentially.
    hpx::generate(hpx::execution::seq, std::begin(v), std::end(v), &std::rand);

    // We can launch the sort in parallel and asynchronously.
    hpx::future<void> done_sorting =
        hpx::sort(hpx::execution::par(          // In parallel.
                      hpx::execution::task),    // Asynchronously.
```

(continues on next page)

(continued from previous page)

```
            std::begin(v), std::end(v));

    // We launch the final task when the vector has been sorted and result is
    // ready using when_all.
    auto all = hpx::when_all(result, done_sorting).then(&final_task);

    // We can wait for all to be ready.
    all.wait();

    // all must be ready at this point because we waited for it to be ready.
    std::cout << (all.is_ready() ? "all is ready!" : "all is not ready...")
              << std::endl;

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

Try copying the contents to your `main.cpp` file and look at the output. It can be a good idea to go through the program step by step with a debugger. You can also try changing the types or adding new arguments to functions to make sure you can get the types to match. The type of the `then` method can be especially tricky to get right (the continuation needs to take the future as an argument).

Note: *HPX* programs accept command line arguments. The most important one is `--hpx:threads=N` to set the number of OS threads used by *HPX*. *HPX* uses one thread per core by default. Play around with the example above and see what difference the number of threads makes on the `sort` function. See *Launching and configuring HPX applications* for more details on how and what options you can pass to *HPX*.

Tip: The example above used the construction `hpx::when_all(...).then(...)`. For convenience and performance it is a good idea to replace uses of `hpx::when_all(...).then(...)` with `dataflow`. See *Dataflow* for more details on `dataflow`.

Tip: If possible, try to use the provided parallel algorithms instead of writing your own implementation. This can save you time and the resulting program is often faster.

2.1.4 Next steps

If you haven't done so already, reading the *Terminology* section will help you get familiar with the terms used in *HPX*.

The *Examples* section contains small, self-contained walkthroughs of example *HPX* programs. The *Local to remote* example is a thorough, realistic example starting from a single node implementation and going stepwise to a distributed implementation.

The *Manual* contains detailed information on writing, building and running *HPX* applications.

2.2 Examples

The following sections analyze some examples to help you get familiar with the *HPX* style of programming. We start off with simple examples that utilize basic *HPX* elements and then begin to expose the reader to the more complex and powerful *HPX* concepts. Section *Building tests and examples* shows how you can build the examples.

2.2.1 Asynchronous execution

The Fibonacci sequence is a sequence of numbers starting with 0 and 1 where every subsequent number is the sum of the previous two numbers. In this example, we will use *HPX* to calculate the value of the n-th element of the Fibonacci sequence. In order to compute this problem in parallel, we will use a facility known as a future.

As shown in the Fig. 2.1 below, a future encapsulates a delayed computation. It acts as a proxy for a result initially not known, most of the time because the computation of the result has not completed yet. The future synchronizes the access of this value by optionally suspending any *HPX*-threads requesting the result until the value is available. When a future is created, it spawns a new *HPX*-thread (either remotely with a *parcel* or locally by placing it into the thread queue) which, when run, will execute the function associated with the future. The arguments of the function are bound when the future is created.

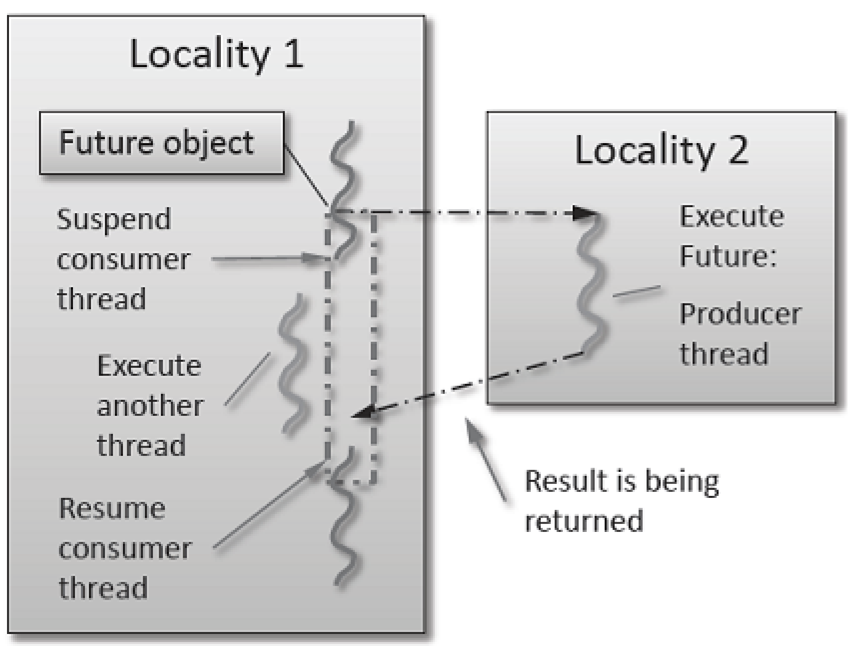

Fig. 2.1: Schematic of a future execution.

Once the function has finished executing, a write operation is performed on the future. The write operation marks the future as completed, and optionally stores data returned by the function. When the result of the delayed computation is needed, a read operation is performed on the future. If the future's function hasn't completed when a read operation is performed on it, the reader *HPX*-thread is suspended until the future is ready. The future facility allows *HPX* to schedule work early in a program so that when the function value is needed it will already be calculated and available. We use this property in our Fibonacci example below to enable its parallel execution.

Setup

The source code for this example can be found here: `fibonacci_local.cpp`.

To compile this program, go to your *HPX* build directory (see *Building HPX* for information on configuring and building *HPX*) and enter:

```
$ make examples.quickstart.fibonacci_local
```

To run the program type:

```
$ ./bin/fibonacci_local
```

This should print (time should be approximate):

```
fibonacci(10) == 55
elapsed time: 0.002430 [s]
```

This run used the default settings, which calculate the tenth element of the Fibonacci sequence. To declare which Fibonacci value you want to calculate, use the `--n-value` option. Additionally you can use the *--hpx:threads* option to declare how many OS-threads you wish to use when running the program. For instance, running:

```
$ ./bin/fibonacci --n-value 20 --hpx:threads 4
```

Will yield:

```
fibonacci(20) == 6765
elapsed time: 0.062854 [s]
```

Walkthrough

Now that you have compiled and run the code, let's look at how the code works. Since this code is written in C++, we will begin with the `main()` function. Here you can see that in *HPX*, `main()` is only used to initialize the runtime system. It is important to note that application-specific command line options are defined here. *HPX* uses Boost.Program_options[7] for command line processing. You can see that our programs `--n-value` option is set by calling the `add_options()` method on an instance of `hpx::program_options::options_description`. The default value of the variable is set to 10. This is why when we ran the program for the first time without using the `--n-value` option the program returned the 10th value of the Fibonacci sequence. The constructor argument of the description is the text that appears when a user uses the *--hpx:help* option to see what command line options are available. `HPX_APPLICATION_STRING` is a macro that expands to a string constant containing the name of the *HPX* application currently being compiled.

In *HPX* `main()` is used to initialize the runtime system and pass the command line arguments to the program. If you wish to add command line options to your program you would add them here using the instance of the Boost class `options_description`, and invoking the public member function `.add_options()` (see Boost Documentation[8] for more details). *hpx::init* calls `hpx_main()` after setting up *HPX*, which is where the logic of our program is encoded.

[7] https://www.boost.org/doc/html/program_options.html
[8] https://www.boost.org/doc/

```
int main(int argc, char* argv[])
{
    // Configure application-specific options
    hpx::program_options::options_description desc_commandline(
        "Usage: " HPX_APPLICATION_STRING " [options]");

    // clang-format off
    desc_commandline.add_options()
        ("n-value",
            hpx::program_options::value<std::uint64_t>()->default_value(10),
            "n value for the Fibonacci function")
        ;
    // clang-format on

    // Initialize and run HPX
    hpx::local::init_params init_args;
    init_args.desc_cmdline = desc_commandline;

    return hpx::local::init(hpx_main, argc, argv, init_args);
}
```

The *hpx::init* function in main() starts the runtime system, and invokes hpx_main() as the first *HPX*-thread. Below we can see that the basic program is simple. The command line option --n-value is read in, a timer (*hpx::chrono::high_resolution_timer*) is set up to record the time it takes to do the computation, the fibonacci function is invoked synchronously, and the answer is printed out.

```
int hpx_main(hpx::program_options::variables_map& vm)
{
    hpx::threads::add_scheduler_mode(
        hpx::threads::policies::scheduler_mode::fast_idle_mode);

    // extract command line argument, i.e. fib(N)
    std::uint64_t n = vm["n-value"].as<std::uint64_t>();

    {
        // Keep track of the time required to execute.
        hpx::chrono::high_resolution_timer t;

        std::uint64_t r = fibonacci(n);

        char const* fmt = "fibonacci({1}) == {2}\nelapsed time: {3} [s]\n";
        hpx::util::format_to(std::cout, fmt, n, r, t.elapsed());
    }

    return hpx::local::finalize();    // Handles HPX shutdown
}
```

The fibonacci function itself is synchronous as the work done inside is asynchronous. To understand what is happening we have to look inside the fibonacci function:

```
std::uint64_t fibonacci(std::uint64_t n)
{
    if (n < 2)
```

(continues on next page)

(continued from previous page)

```
        return n;

    hpx::future<std::uint64_t> n1 = hpx::async(fibonacci, n - 1);
    std::uint64_t n2 = fibonacci(n - 2);

    return n1.get() + n2;    // wait for the Future to return their values
}
```

This block of code looks similar to regular C++ code. First, `if (n < 2)`, meaning n is 0 or 1, then we return 0 or 1 (recall the first element of the Fibonacci sequence is 0 and the second is 1). If n is larger than 1 we spawn two new tasks whose results are contained in `n1` and `n2`. This is done using *`hpx::async`* which takes as arguments a function (function pointer, object or lambda) and the arguments to the function. Instead of returning a `std::uint64_t` like `fibonacci` does, `hpx::async` returns a future of a `std::uint64_t`, i.e. `hpx::future<std::uint64_t>`. Each of these futures represents an asynchronous, recursive call to `fibonacci`. After we've created the futures, we wait for both of them to finish computing, we add them together, and return that value as our result. We get the values from the futures using the `get` method. The recursive call tree will continue until n is equal to 0 or 1, at which point the value can be returned because it is implicitly known. When this termination condition is reached, the futures can then be added up, producing the n-th value of the Fibonacci sequence.

Note that calling `get` potentially blocks the calling *HPX*-thread, and lets other *HPX*-threads run in the meantime. There are, however, more efficient ways of doing this. `examples/quickstart/fibonacci_futures.cpp` contains many more variations of locally computing the Fibonacci numbers, where each method makes different tradeoffs in where asynchrony and parallelism is applied. To get started, however, the method above is sufficient and optimizations can be applied once you are more familiar with *HPX*. The example *Dataflow* presents dataflow, which is a way to more efficiently chain together multiple tasks.

2.2.2 Parallel algorithms

This program will perform a matrix multiplication in parallel. The output will look something like this:

```
Matrix A is :
4 9 6
1 9 8

Matrix B is :
4 9
6 1
9 8

Resultant Matrix is :
124 93
130 82
```

Setup

The source code for this example can be found here: `matrix_multiplication.cpp`.

To compile this program, go to your *HPX* build directory (see *Building HPX* for information on configuring and building *HPX*) and enter:

```
$ make examples.quickstart.matrix_multiplication
```

To run the program type:

```
$ ./bin/matrix_multiplication
```

or:

```
$ ./bin/matrix_multiplication --n 2 --m 3 --k 2 --s 100 --l 0 --u 10
```

where the first matrix is *n* x *m* and the second *m* x *k*, s is the seed for creating the random values of the matrices and the range of these values is [l,u]

This should print:

```
Matrix A is :
4 9 6
1 9 8

Matrix B is :
4 9
6 1
9 8

Resultant Matrix is :
124 93
130 82
```

Notice that the numbers may be different because of the random initialization of the matrices.

Walkthrough

Now that you have compiled and run the code, let's look at how the code works.

First, `main()` is used to initialize the runtime system and pass the command line arguments to the program. `hpx::init` calls `hpx_main()` after setting up HPX, which is where our program is implemented.

```
int main(int argc, char* argv[])
{
    using namespace hpx::program_options;
    options_description cmdline("usage: " HPX_APPLICATION_STRING " [options]");
    // clang-format off
    cmdline.add_options()
        ("n",
        hpx::program_options::value<std::size_t>()->default_value(2),
        "Number of rows of first matrix")
        ("m",
        hpx::program_options::value<std::size_t>()->default_value(3),
```

(continues on next page)

(continued from previous page)

```
        "Number of columns of first matrix (equal to the number of rows of "
        "second matrix)")
        ("k",
        hpx::program_options::value<std::size_t>()->default_value(2),
        "Number of columns of second matrix")
        ("seed,s",
        hpx::program_options::value<unsigned int>(),
        "The random number generator seed to use for this run")
        ("l",
        hpx::program_options::value<int>()->default_value(0),
        "Lower limit of range of values")
        ("u",
        hpx::program_options::value<int>()->default_value(10),
        "Upper limit of range of values");
    // clang-format on
    hpx::local::init_params init_args;
    init_args.desc_cmdline = cmdline;

    return hpx::local::init(hpx_main, argc, argv, init_args);
}
```

Proceeding to the `hpx_main()` function, we can see that matrix multiplication can be done very easily.

```
int hpx_main(hpx::program_options::variables_map& vm)
{
    using element_type = int;

    // Define matrix sizes
    std::size_t const rowsA = vm["n"].as<std::size_t>();
    std::size_t const colsA = vm["m"].as<std::size_t>();
    std::size_t const rowsB = colsA;
    std::size_t const colsB = vm["k"].as<std::size_t>();
    std::size_t const rowsR = rowsA;
    std::size_t const colsR = colsB;

    // Initialize matrices A and B
    std::vector<int> A(rowsA * colsA);
    std::vector<int> B(rowsB * colsB);
    std::vector<int> R(rowsR * colsR);

    // Define seed
    unsigned int seed = std::random_device{}();
    if (vm.count("seed"))
        seed = vm["seed"].as<unsigned int>();

    gen.seed(seed);
    std::cout << "using seed: " << seed << std::endl;

    // Define range of values
    int const lower = vm["l"].as<int>();
    int const upper = vm["u"].as<int>();
```

(continues on next page)

(continued from previous page)

```
    // Matrices have random values in the range [lower, upper]
    std::uniform_int_distribution<element_type> dis(lower, upper);
    auto generator = std::bind(dis, gen);
    hpx::ranges::generate(A, generator);
    hpx::ranges::generate(B, generator);

    // Perform matrix multiplication
    hpx::experimental::for_loop(hpx::execution::par, 0, rowsA, [&](auto i) {
        hpx::experimental::for_loop(0, colsB, [&](auto j) {
            R[i * colsR + j] = 0;
            hpx::experimental::for_loop(0, rowsB, [&](auto k) {
                R[i * colsR + j] += A[i * colsA + k] * B[k * colsB + j];
            });
        });
    });

    // Print all 3 matrices
    print_matrix(A, rowsA, colsA, "A");
    print_matrix(B, rowsB, colsB, "B");
    print_matrix(R, rowsR, colsR, "R");

    return hpx::local::finalize();
}
```

First, the dimensions of the matrices are defined. If they were not given as command-line arguments, their default values are *2* x *3* for the first matrix and *3* x *2* for the second. We use standard vectors to define the matrices to be multiplied as well as the resultant matrix.

To give some random initial values to our matrices, we use std::uniform_int_distribution[9]. Then, `std::bind()` is used along with `hpx::ranges::generate()` to yield two matrices A and B, which contain values in the range of [0, 10] or in the range defined by the user at the command-line arguments. The seed to generate the values can also be defined by the user.

The next step is to perform the matrix multiplication in parallel. This can be done by just using an *hpx::experimental::for_loop* combined with a parallel execution policy `hpx::execution::par` as the outer loop of the multiplication. Note that the execution of *hpx::experimental::for_loop* without specifying an execution policy is equivalent to specifying `hpx::execution::seq` as the execution policy.

Finally, the matrices A, B that are multiplied as well as the resultant matrix R are printed using the following function.

```
void print_matrix(std::vector<int> const& M, std::size_t rows, std::size_t cols,
    char const* message)
{
    std::cout << "\nMatrix " << message << " is:" << std::endl;
    for (std::size_t i = 0; i < rows; i++)
    {
        for (std::size_t j = 0; j < cols; j++)
            std::cout << M[i * cols + j] << " ";
        std::cout << "\n";
    }
}
```

[9] https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

2.2.3 Asynchronous execution with actions

This example extends the *previous example* by introducing *actions*: functions that can be run remotely. In this example, however, we will still only run the action locally. The mechanism to execute *actions* stays the same: `hpx::async`. Later examples will demonstrate running actions on remote *localities* (e.g. *Remote execution with actions*).

Setup

The source code for this example can be found here: `fibonacci.cpp`.

To compile this program, go to your *HPX* build directory (see *Building HPX* for information on configuring and building *HPX*) and enter:

```
$ make examples.quickstart.fibonacci
```

To run the program type:

```
$ ./bin/fibonacci
```

This should print (time should be approximate):

```
fibonacci(10) == 55
elapsed time: 0.00186288 [s]
```

This run used the default settings, which calculate the tenth element of the Fibonacci sequence. To declare which Fibonacci value you want to calculate, use the `--n-value` option. Additionally you can use the `--hpx:threads` option to declare how many OS-threads you wish to use when running the program. For instance, running:

```
$ ./bin/fibonacci --n-value 20 --hpx:threads 4
```

Will yield:

```
fibonacci(20) == 6765
elapsed time: 0.233827 [s]
```

Walkthrough

The code needed to initialize the *HPX* runtime is the same as in the *previous example*:

```
int main(int argc, char* argv[])
{
    // Configure application-specific options
    hpx::program_options::options_description desc_commandline(
        "Usage: " HPX_APPLICATION_STRING " [options]");

    desc_commandline.add_options()("n-value",
        hpx::program_options::value<std::uint64_t>()->default_value(10),
        "n value for the Fibonacci function");

    // Initialize and run HPX
    hpx::init_params init_args;
    init_args.desc_cmdline = desc_commandline;
```

(continues on next page)

(continued from previous page)

```
    return hpx::init(argc, argv, init_args);
}
```

The *hpx::init* function in `main()` starts the runtime system, and invokes `hpx_main()` as the first *HPX*-thread. The command line option `--n-value` is read in, a timer (*hpx::chrono::high_resolution_timer*) is set up to record the time it takes to do the computation, the `fibonacci` *action* is invoked synchronously, and the answer is printed out.

```
int hpx_main(hpx::program_options::variables_map& vm)
{
    // extract command line argument, i.e. fib(N)
    std::uint64_t n = vm["n-value"].as<std::uint64_t>();

    {
        // Keep track of the time required to execute.
        hpx::chrono::high_resolution_timer t;

        // Wait for fib() to return the value
        fibonacci_action fib;
        std::uint64_t r = fib(hpx::find_here(), n);

        char const* fmt = "fibonacci({1}) == {2}\nelapsed time: {3} [s]\n";
        hpx::util::format_to(std::cout, fmt, n, r, t.elapsed());
    }

    return hpx::finalize();    // Handles HPX shutdown
}
```

Upon a closer look we see that we've created a `std::uint64_t` to store the result of invoking our `fibonacci_action` `fib`. This *action* will launch synchronously (as the work done inside of the *action* will be asynchronous itself) and return the result of the Fibonacci sequence. But wait, what is an *action*? And what is this `fibonacci_action`? For starters, an *action* is a wrapper for a function. By wrapping functions, *HPX* can send packets of work to different processing units. These vehicles allow users to calculate work now, later, or on certain nodes. The first argument to our *action* is the location where the *action* should be run. In this case, we just want to run the *action* on the machine that we are currently on, so we use *hpx::find_here*. To further understand this we turn to the code to find where `fibonacci_action` was defined:

```
// forward declaration of the Fibonacci function
std::uint64_t fibonacci(std::uint64_t n);

// This is to generate the required boilerplate we need for the remote
// invocation to work.
HPX_PLAIN_ACTION(fibonacci, fibonacci_action)
```

A plain *action* is the most basic form of *action*. Plain *action*s wrap simple global functions which are not associated with any particular object (we will discuss other types of *action*s in *Components and actions*). In this block of code the function `fibonacci()` is declared. After the declaration, the function is wrapped in an *action* in the declaration *HPX_PLAIN_ACTION*. This function takes two arguments: the name of the function that is to be wrapped and the name of the *action* that you are creating.

This picture should now start making sense. The function `fibonacci()` is wrapped in an *action* `fibonacci_action`, which was run synchronously but created asynchronous work, then returns a `std::uint64_t` representing the result of the function `fibonacci()`. Now, let's look at the function `fibonacci()`:

```
std::uint64_t fibonacci(std::uint64_t n)
{
    if (n < 2)
        return n;

    // We restrict ourselves to execute the Fibonacci function locally.
    hpx::id_type const locality_id = hpx::find_here();

    // Invoking the Fibonacci algorithm twice is inefficient.
    // However, we intentionally demonstrate it this way to create some
    // heavy workload.

    fibonacci_action fib;
    hpx::future<std::uint64_t> n1 = hpx::async(fib, locality_id, n - 1);
    hpx::future<std::uint64_t> n2 = hpx::async(fib, locality_id, n - 2);

    return n1.get() +
        n2.get();    // wait for the Futures to return their values
}
```

This block of code is much more straightforward and should look familiar from the *previous example*. First, `if (n < 2)`, meaning n is 0 or 1, then we return 0 or 1 (recall the first element of the Fibonacci sequence is 0 and the second is 1). If n is larger than 1 we spawn two tasks using *`hpx::async`*. Each of these futures represents an asynchronous, recursive call to `fibonacci`. As previously we wait for both futures to finish computing, get the results, add them together, and return that value as our result. The recursive call tree will continue until n is equal to 0 or 1, at which point the value can be returned because it is implicitly known. When this termination condition is reached, the futures can then be added up, producing the n-th value of the Fibonacci sequence.

2.2.4 Remote execution with actions

This program will print out a hello world message on every OS-thread on every *locality*. The output will look something like this:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 1 on locality 1
hello world from OS-thread 0 on locality 0
hello world from OS-thread 0 on locality 1
```

Setup

The source code for this example can be found here: `hello_world_distributed.cpp`.

To compile this program, go to your *HPX* build directory (see *Building HPX* for information on configuring and building *HPX*) and enter:

```
$ make examples.quickstart.hello_world_distributed
```

To run the program type:

```
$ ./bin/hello_world_distributed
```

This should print:

```
hello world from OS-thread 0 on locality 0
```

To use more OS-threads use the command line option *--hpx:threads* and type the number of threads that you wish to use. For example, typing:

```
$ ./bin/hello_world_distributed --hpx:threads 2
```

will yield:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
```

Notice how the ordering of the two print statements will change with subsequent runs. To run this program on multiple localities please see the section *How to use HPX applications with PBS*.

Walkthrough

Now that you have compiled and run the code, let's look at how the code works, beginning with `main()`:

```
// Here is the main entry point. By using the include 'hpx/hpx_main.hpp' HPX
// will invoke the plain old C-main() as its first HPX thread.
int main()
{
    // Get a list of all available localities.
    std::vector<hpx::id_type> localities = hpx::find_all_localities();

    // Reserve storage space for futures, one for each locality.
    std::vector<hpx::future<void>> futures;
    futures.reserve(localities.size());

    for (hpx::id_type const& node : localities)
    {
        // Asynchronously start a new task. The task is encapsulated in a
        // future, which we can query to determine if the task has
        // completed.
        typedef hello_world_foreman_action action_type;
        futures.push_back(hpx::async<action_type>(node));
    }

    // The non-callback version of hpx::wait_all takes a single parameter,
    // a vector of futures to wait on. hpx::wait_all only returns when
    // all of the futures have finished.
    hpx::wait_all(futures);
    return 0;
}
```

In this excerpt of the code we again see the use of futures. This time the futures are stored in a vector so that they can easily be accessed. *hpx::wait_all* is a family of functions that wait on for an `std::vector<>` of futures to become ready. In this piece of code, we are using the synchronous version of *hpx::wait_all*, which takes one argument (the `std::vector<>` of futures to wait on). This function will not return until all the futures in the vector have been executed.

In *Asynchronous execution with actions* we used *hpx::find_here* to specify the target of our actions. Here, we instead use *hpx::find_all_localities*, which returns an `std::vector<>` containing the identifiers of all the machines in the system, including the one that we are on.

As in *Asynchronous execution with actions* our futures are set using *hpx::async<>*. The `hello_world_foreman_action` is declared here:

```
// Define the boilerplate code necessary for the function 'hello_world_foreman'
// to be invoked as an HPX action.
HPX_PLAIN_ACTION(hello_world_foreman, hello_world_foreman_action)
```

Another way of thinking about this wrapping technique is as follows: functions (the work to be done) are wrapped in actions, and actions can be executed locally or remotely (e.g. on another machine participating in the computation).

Now it is time to look at the `hello_world_foreman()` function which was wrapped in the action above:

```
void hello_world_foreman()
{
    // Get the number of worker OS-threads in use by this locality.
    std::size_t const os_threads = hpx::get_os_thread_count();

    // Populate a set with the OS-thread numbers of all OS-threads on this
    // locality. When the hello world message has been printed on a particular
    // OS-thread, we will remove it from the set.
    std::set<std::size_t> attendance;
    for (std::size_t os_thread = 0; os_thread < os_threads; ++os_thread)
        attendance.insert(os_thread);

    // As long as there are still elements in the set, we must keep scheduling
    // HPX-threads. Because HPX features work-stealing task schedulers, we have
    // no way of enforcing which worker OS-thread will actually execute
    // each HPX-thread.
    while (!attendance.empty())
    {
        // Each iteration, we create a task for each element in the set of
        // OS-threads that have not said "Hello world". Each of these tasks
        // is encapsulated in a future.
        std::vector<hpx::future<std::size_t>> futures;
        futures.reserve(attendance.size());

        for (std::size_t worker : attendance)
        {
            // Asynchronously start a new task. The task is encapsulated in a
            // future that we can query to determine if the task has completed.
            //
            // We give the task a hint to run on a particular worker thread
            // (core) and suggest binding the scheduled thread to the given
            // core, but no guarantees are given by the scheduler that the task
            // will actually run on that worker thread. It will however try as
            // hard as possible to place the new task on the given worker
            // thread.
            hpx::execution::parallel_executor exec(
                hpx::threads::thread_priority::bound);

            hpx::threads::thread_schedule_hint hint(
                hpx::threads::thread_schedule_hint_mode::thread,
                static_cast<std::int16_t>(worker));
```

(continues on next page)

(continued from previous page)

```
            futures.push_back(
                hpx::async(hpx::execution::experimental::with_hint(exec, hint),
                    hello_world_worker, worker));
        }

        // Wait for all of the futures to finish. The callback version of the
        // hpx::wait_each function takes two arguments: a vector of futures,
        // and a binary callback.  The callback takes two arguments; the first
        // is the index of the future in the vector, and the second is the
        // return value of the future. hpx::wait_each doesn't return until
        // all the futures in the vector have returned.
        hpx::spinlock mtx;
        hpx::wait_each(hpx::unwrapping([&](std::size_t t) {
            if (std::size_t(-1) != t)
            {
                std::lock_guard<hpx::spinlock> lk(mtx);
                attendance.erase(t);
            }
        }),
            futures);
    }
}
```

Now, before we discuss `hello_world_foreman()`, let's talk about the *`hpx::wait_each`* function. The version of *`hpx::wait_each`* invokes a callback function provided by the user, supplying the callback function with the result of the future.

In `hello_world_foreman()`, an `std::set<>` called `attendance` keeps track of which OS-threads have printed out the hello world message. When the OS-thread prints out the statement, the future is marked as ready, and *`hpx::wait_each`* in `hello_world_foreman()`. If it is not executing on the correct OS-thread, it returns a value of -1, which causes `hello_world_foreman()` to leave the OS-thread id in `attendance`.

```
std::size_t hello_world_worker(std::size_t desired)
{
    // Returns the OS-thread number of the worker that is running this
    // HPX-thread.
    std::size_t current = hpx::get_worker_thread_num();
    if (current == desired)
    {
        // The HPX-thread has been run on the desired OS-thread.
        char const* msg = "hello world from OS-thread {1} on locality {2}\n";

        hpx::util::format_to(hpx::cout, msg, desired, hpx::get_locality_id())
            << std::flush;

        return desired;
    }

    // This HPX-thread has been run by the wrong OS-thread, make the foreman
    // try again by rescheduling it.
    return std::size_t(-1);
}
```

Because *HPX* features work stealing task schedulers, there is no way to guarantee that an action will be scheduled on a particular OS-thread. This is why we must use a guess-and-check approach.

2.2.5 Components and actions

The accumulator example demonstrates the use of components. Components are C++ classes that expose methods as a type of *HPX* action. These actions are called component actions.

Components are globally named, meaning that a component action can be called remotely (e.g., from another machine). There are two accumulator examples in *HPX*.

In the *Asynchronous execution with actions* and the *Remote execution with actions*, we introduced plain actions, which wrapped global functions. The target of a plain action is an identifier which refers to a particular machine involved in the computation. For plain actions, the target is the machine where the action will be executed.

Component actions, however, do not target machines. Instead, they target component instances. The instance may live on the machine that we've invoked the component action from, or it may live on another machine.

The component in this example exposes three different functions:

- `reset()` - Resets the accumulator value to 0.
- `add(arg)` - Adds `arg` to the accumulators value.
- `query()` - Queries the value of the accumulator.

This example creates an instance of the accumulator, and then allows the user to enter commands at a prompt, which subsequently invoke actions on the accumulator instance.

Setup

The source code for this example can be found here: `accumulator_client.cpp`.

To compile this program, go to your *HPX* build directory (see *Building HPX* for information on configuring and building *HPX*) and enter:

```
$ make examples.accumulators.accumulator
```

To run the program type:

```
$ ./bin/accumulator_client
```

Once the program starts running, it will print the following prompt and then wait for input. An example session is given below:

```
commands: reset, add [amount], query, help, quit
> add 5
> add 10
> query
15
> add 2
> query
17
> reset
> add 1
> query
```

(continues on next page)

(continued from previous page)

```
1
> quit
```

Walkthrough

Now, let's take a look at the source code of the accumulator example. This example consists of two parts: an *HPX* component library (a library that exposes an *HPX* component) and a client application which uses the library. This walkthrough will cover the *HPX* component library. The code for the client application can be found here: `accumulator_client.cpp`.

An *HPX* component is represented by two C++ classes:

- **A server class** - The implementation of the component's functionality.
- **A client class** - A high-level interface that acts as a proxy for an instance of the component.

Typically, these two classes both have the same name, but the server class usually lives in different sub-namespaces (`server`). For example, the full names of the two classes in accumulator are:

- `examples::server::accumulator` (server class)
- `examples::accumulator` (client class)

The server class

The following code is from: `accumulator.hpp`.

All *HPX* component server classes must inherit publicly from the *HPX* component base class: `hpx::components::component_base`

The accumulator component inherits from `hpx::components::locking_hook`. This allows the runtime system to ensure that all action invocations are serialized. That means that the system ensures that no two actions are invoked at the same time on a given component instance. This makes the component thread safe and no additional locking has to be implemented by the user. Moreover, an accumulator component is a component because it also inherits from `hpx::components::component_base` (the template argument passed to locking_hook is used as its base class). The following snippet shows the corresponding code:

```
    class accumulator
      : public hpx::components::locking_hook<
            hpx::components::component_base<accumulator>>
```

Our accumulator class will need a data member to store its value in, so let's declare a data member:

```
        argument_type value_;
```

The constructor for this class simply initializes `value_` to 0:

```
        accumulator()
          : value_(0)
        {
        }
```

Next, let's look at the three methods of this component that we will be exposing as component actions:

Here are the action types. These types wrap the methods we're exposing. The wrapping technique is very similar to the one used in the *Asynchronous execution with actions* and the *Remote execution with actions*:

```
        HPX_DEFINE_COMPONENT_ACTION(accumulator, reset)
        HPX_DEFINE_COMPONENT_ACTION(accumulator, add)
        HPX_DEFINE_COMPONENT_ACTION(accumulator, query)
```

The last piece of code in the server class header is the declaration of the action type registration code:

```
HPX_REGISTER_ACTION_DECLARATION(
    examples::server::accumulator::reset_action, accumulator_reset_action)

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::accumulator::add_action, accumulator_add_action)

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::accumulator::query_action, accumulator_query_action)
```

Note: The code above must be placed in the global namespace.

The rest of the registration code is in `accumulator.cpp`

```
///////////////////////////////////////////////////////////////////////////////
// Add factory registration functionality.
HPX_REGISTER_COMPONENT_MODULE()

///////////////////////////////////////////////////////////////////////////////
typedef hpx::components::component<examples::server::accumulator>
    accumulator_type;

HPX_REGISTER_COMPONENT(accumulator_type, accumulator)

///////////////////////////////////////////////////////////////////////////////
// Serialization support for accumulator actions.
HPX_REGISTER_ACTION(
    accumulator_type::wrapped_type::reset_action, accumulator_reset_action)
HPX_REGISTER_ACTION(
    accumulator_type::wrapped_type::add_action, accumulator_add_action)
HPX_REGISTER_ACTION(
    accumulator_type::wrapped_type::query_action, accumulator_query_action)
```

Note: The code above must be placed in the global namespace.

The client class

The following code is from accumulator.hpp.

The client class is the primary interface to a component instance. Client classes are used to create components:

```
// Create a component on this locality.
examples::accumulator c = hpx::new_<examples::accumulator>(hpx::find_here());
```

and to invoke component actions:

```
c.add(hpx::launch::apply, 4);
```

Clients, like servers, need to inherit from a base class, this time, `hpx::components::client_base`:

```
    class accumulator
      : public hpx::components::client_base<accumulator, server::accumulator>
```

For readability, we typedef the base class like so:

```
        typedef hpx::components::client_base<accumulator, server::accumulator>
            base_type;
```

Here are examples of how to expose actions through a client class:

There are a few different ways of invoking actions:

- **Non-blocking**: For actions that don't have return types, or when we do not care about the result of an action, we can invoke the action using fire-and-forget semantics. This means that once we have asked *HPX* to compute the action, we forget about it completely and continue with our computation. We use *hpx::post* to invoke an action in a non-blocking fashion.

```
        void reset(hpx::launch::apply_policy)
        {
            HPX_ASSERT(this->get_id());

            typedef server::accumulator::reset_action action_type;
            hpx::post<action_type>(this->get_id());
        }
```

- **Asynchronous**: Futures, as demonstrated in *Asynchronous execution*, *Asynchronous execution with actions*, and the *Remote execution with actions*, enable asynchronous action invocation. Here's an example from the accumulator client class:

```
        hpx::future<argument_type> query(hpx::launch::async_policy)
        {
            HPX_ASSERT(this->get_id());

            typedef server::accumulator::query_action action_type;
            return hpx::async<action_type>(hpx::launch::async, this->get_id());
        }
```

- **Synchronous**: To invoke an action in a fully synchronous manner, we can simply call *hpx::async*().get() (i.e., create a future and immediately wait on it to be ready). Here's an example from the accumulator client class:

```
        void add(argument_type arg)
        {
            HPX_ASSERT(this->get_id());

            typedef server::accumulator::add_action action_type;
            action_type()(this->get_id(), arg);
        }
```

Note that `this->get_id()` references a data member of the `hpx::components::client_base` base class which identifies the server accumulator instance.

`hpx::naming::id_type` is a type which represents a global identifier in *HPX*. This type specifies the target of an action. This is the type that is returned by *`hpx::find_here`* in which case it represents the *locality* the code is running on.

2.2.6 Dataflow

HPX provides its users with several different tools to simply express parallel concepts. One of these tools is a *local control object* (*LCO*) called dataflow. An *LCO* is a type of component that can spawn a new thread when triggered. They are also distinguished from other components by a standard interface that allow users to understand and use them easily. A Dataflow, being an *LCO*, is triggered when the values it depends on become available. For instance, if you have a calculation X that depends on the results of three other calculations, you could set up a dataflow that would begin the calculation X as soon as the other three calculations have returned their values. Dataflows are set up to depend on other dataflows. It is this property that makes dataflow a powerful parallelization tool. If you understand the dependencies of your calculation, you can devise a simple algorithm that sets up a dependency tree to be executed. In this example, we calculate compound interest. To calculate compound interest, one must calculate the interest made in each compound period, and then add that interest back to the principal before calculating the interest made in the next period. A practical person would, of course, use the formula for compound interest:

$$F = P(1 + i)^n$$

where F is the future value, P is the principal value, i is the interest rate, and n is the number of compound periods.

However, for the sake of this example, we have chosen to manually calculate the future value by iterating:

$$I = Pi$$

and

$$P = P + I$$

Setup

The source code for this example can be found here: `interest_calculator.cpp`.

To compile this program, go to your *HPX* build directory (see *Building HPX* for information on configuring and building *HPX*) and enter:

```
$ make examples.quickstart.interest_calculator
```

To run the program type:

```
$ ./bin/interest_calculator --principal 100 --rate 5 --cp 6 --time 36
Final amount: 134.01
Amount made: 34.0096
```

Walkthrough

Let us begin with main. Here we can see that we again are using Boost.Program_options to set our command line variables (see *Asynchronous execution with actions* for more details). These options set the principal, rate, compound period, and time. It is important to note that the units of time for `cp` and `time` must be the same.

```
int main(int argc, char** argv)
{
    options_description cmdline("Usage: " HPX_APPLICATION_STRING " [options]");

    cmdline.add_options()("principal", value<double>()->default_value(1000),
        "The principal [$]")("rate", value<double>()->default_value(7),
        "The interest rate [%]")("cp", value<int>()->default_value(12),
        "The compound period [months]")("time",
        value<int>()->default_value(12 * 30),
        "The time money is invested [months]");

    hpx::init_params init_args;
    init_args.desc_cmdline = cmdline;

    return hpx::init(argc, argv, init_args);
}
```

Next we look at hpx_main.

```
int hpx_main(variables_map& vm)
{
    {
        using hpx::dataflow;
        using hpx::make_ready_future;
        using hpx::shared_future;
        using hpx::unwrapping;
        hpx::id_type here = hpx::find_here();

        double init_principal =
            vm["principal"].as<double>();              //Initial principal
        double init_rate = vm["rate"].as<double>();    //Interest rate
        int cp = vm["cp"].as<int>();      //Length of a compound period
        int t = vm["time"].as<int>();     //Length of time money is invested

        init_rate /= 100;     //Rate is a % and must be converted
        t /= cp;     //Determine how many times to iterate interest calculation:
            //How many full compound periods can fit in the time invested

        // In non-dataflow terms the implemented algorithm would look like:
        //
        // int t = 5;     // number of time periods to use
        // double principal = init_principal;
        // double rate = init_rate;
        //
        // for (int i = 0; i < t; ++i)
        // {
        //     double interest = calc(principal, rate);
        //     principal = add(principal, interest);
```

(continues on next page)

(continued from previous page)

```
        // }
        //
        // Please note the similarity with the code below!

        shared_future<double> principal = make_ready_future(init_principal);
        shared_future<double> rate = make_ready_future(init_rate);

        for (int i = 0; i < t; ++i)
        {
            shared_future<double> interest =
                dataflow(unwrapping(calc), principal, rate);
            principal = dataflow(unwrapping(add), principal, interest);
        }

        // wait for the dataflow execution graph to be finished calculating our
        // overall interest
        double result = principal.get();

        std::cout << "Final amount: " << result << std::endl;
        std::cout << "Amount made: " << result - init_principal << std::endl;
    }

    return hpx::finalize();
}
```

Here we find our command line variables read in, the rate is converted from a percent to a decimal, the number of calculation iterations is determined, and then our shared_futures are set up. Notice that we first place our principal and rate into shares futures by passing the variables `init_principal` and `init_rate` using *hpx::make_ready_future*.

In this way *hpx::shared_future*<double> `principal` and `rate` will be initialized to `init_principal` and `init_rate` when *hpx::make_ready_future*<double> returns a future containing those initial values. These shared futures then enter the for loop and are passed to `interest`. Next `principal` and `interest` are passed to the reassignment of `principal` using a *hpx::dataflow*. A dataflow will first wait for its arguments to be ready before launching any callbacks, so `add` in this case will not begin until both `principal` and `interest` are ready. This loop continues for each compound period that must be calculated. To see how `interest` and `principal` are calculated in the loop, let us look at `calc_action` and `add_action`:

```
// Calculate interest for one period
double calc(double principal, double rate)
{
    return principal * rate;
}

///////////////////////////////////////////////////////////////////////////////
// Add the amount made to the principal
double add(double principal, double interest)
{
    return principal + interest;
}
```

After the shared future dependencies have been defined in hpx_main, we see the following statement:

```
double result = principal.get();
```

This statement calls `hpx::future::get` on the shared future principal which had its value calculated by our for loop. The program will wait here until the entire dataflow tree has been calculated and the value assigned to result. The program then prints out the final value of the investment and the amount of interest made by subtracting the final value of the investment from the initial value of the investment.

2.2.7 Local to remote

When developers write code they typically begin with a simple serial code and build upon it until all of the required functionality is present. The following set of examples were developed to demonstrate this iterative process of evolving a simple serial program to an efficient, fully-distributed *HPX* application. For this demonstration, we implemented a 1D heat distribution problem. This calculation simulates the diffusion of heat across a ring from an initialized state to some user-defined point in the future. It does this by breaking each portion of the ring into discrete segments and using the current segment's temperature and the temperature of the surrounding segments to calculate the temperature of the current segment in the next timestep as shown by Fig. 2.2 below.

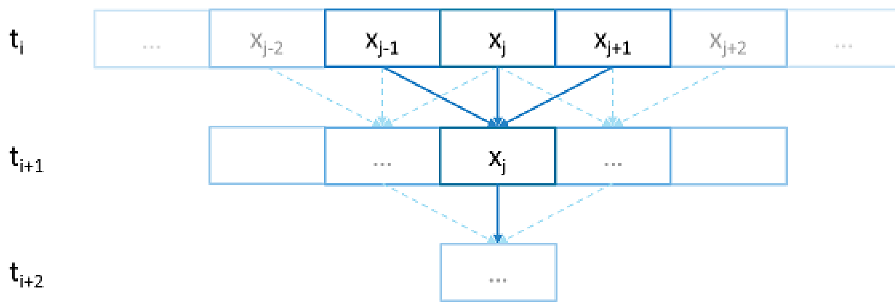

Fig. 2.2: Heat diffusion example program flow.

We parallelize this code over the following eight examples:

- `Example 1`
- `Example 2`
- `Example 3`
- `Example 4`
- `Example 5`
- `Example 6`
- `Example 7`
- `Example 8`

The first example is straight serial code. In this code we instantiate a vector `U` that contains two vectors of doubles as seen in the structure `stepper`.

```
struct stepper
{
    // Our partition type
    typedef double partition;

    // Our data for one time step
```

(continues on next page)

(continued from previous page)

```
    typedef std::vector<partition> space;

    // Our operator
    static double heat(double left, double middle, double right)
    {
        return middle + (k * dt / (dx * dx)) * (left - 2 * middle + right);
    }

    // do all the work on 'nx' data points for 'nt' time steps
    space do_work(std::size_t nx, std::size_t nt)
    {
        // U[t][i] is the state of position i at time t.
        std::vector<space> U(2);
        for (space& s : U)
            s.resize(nx);

        // Initial conditions: f(0, i) = i
        for (std::size_t i = 0; i != nx; ++i)
            U[0][i] = double(i);

        // Actual time step loop
        for (std::size_t t = 0; t != nt; ++t)
        {
            space const& current = U[t % 2];
            space& next = U[(t + 1) % 2];

            next[0] = heat(current[nx - 1], current[0], current[1]);

            for (std::size_t i = 1; i != nx - 1; ++i)
                next[i] = heat(current[i - 1], current[i], current[i + 1]);

            next[nx - 1] = heat(current[nx - 2], current[nx - 1], current[0]);
        }

        // Return the solution at time-step 'nt'.
        return U[nt % 2];
    }
};
```

Each element in the vector of doubles represents a single grid point. To calculate the change in heat distribution, the temperature of each grid point, along with its neighbors, is passed to the function `heat`. In order to improve readability, references named `current` and `next` are created which, depending on the time step, point to the first and second vector of doubles. The first vector of doubles is initialized with a simple heat ramp. After calling the heat function with the data in the `current` vector, the results are placed into the `next` vector.

In example 2 we employ a technique called futurization. Futurization is a method by which we can easily transform a code that is serially executed into a code that creates asynchronous threads. In the simplest case this involves replacing a variable with a future to a variable, a function with a future to a function, and adding a `.get()` at the point where a value is actually needed. The code below shows how this technique was applied to the `struct stepper`.

```
struct stepper
{
    // Our partition type
```

(continues on next page)

(continued from previous page)

```
typedef hpx::shared_future<double> partition;

// Our data for one time step
typedef std::vector<partition> space;

// Our operator
static double heat(double left, double middle, double right)
{
    return middle + (k * dt / (dx * dx)) * (left - 2 * middle + right);
}

// do all the work on 'nx' data points for 'nt' time steps
hpx::future<space> do_work(std::size_t nx, std::size_t nt)
{
    using hpx::dataflow;
    using hpx::unwrapping;

    // U[t][i] is the state of position i at time t.
    std::vector<space> U(2);
    for (space& s : U)
        s.resize(nx);

    // Initial conditions: f(0, i) = i
    for (std::size_t i = 0; i != nx; ++i)
        U[0][i] = hpx::make_ready_future(double(i));

    auto Op = unwrapping(&stepper::heat);

    // Actual time step loop
    for (std::size_t t = 0; t != nt; ++t)
    {
        space const& current = U[t % 2];
        space& next = U[(t + 1) % 2];

        // WHEN U[t][i-1], U[t][i], and U[t][i+1] have been computed, THEN we
        // can compute U[t+1][i]
        for (std::size_t i = 0; i != nx; ++i)
        {
            next[i] =
                dataflow(hpx::launch::async, Op, current[idx(i, -1, nx)],
                    current[i], current[idx(i, +1, nx)]);
        }
    }

    // Now the asynchronous computation is running; the above for-loop does not
    // wait on anything. There is no implicit waiting at the end of each timestep;
    // the computation of each U[t][i] will begin as soon as its dependencies
    // are ready and hardware is available.

    // Return the solution at time-step 'nt'.
    return hpx::when_all(U[nt % 2]);
}
```

(continues on next page)

(continued from previous page)

```
};
```

In example 2, we redefine our partition type as a `shared_future` and, in `main`, create the object `result`, which is a future to a vector of partitions. We use `result` to represent the last vector in a string of vectors created for each timestep. In order to move to the next timestep, the values of a partition and its neighbors must be passed to `heat` once the futures that contain them are ready. In *HPX*, we have an LCO (Local Control Object) named Dataflow that assists the programmer in expressing this dependency. Dataflow allows us to pass the results of a set of futures to a specified function when the futures are ready. Dataflow takes three types of arguments, one which instructs the dataflow on how to perform the function call (async or sync), the function to call (in this case `Op`), and futures to the arguments that will be passed to the function. When called, dataflow immediately returns a future to the result of the specified function. This allows users to string dataflows together and construct an execution tree.

After the values of the futures in dataflow are ready, the values must be pulled out of the future container to be passed to the function `heat`. In order to do this, we use the HPX facility `unwrapping`, which underneath calls `.get()` on each of the futures so that the function `heat` will be passed doubles and not futures to doubles.

By setting up the algorithm this way, the program will be able to execute as quickly as the dependencies of each future are met. Unfortunately, this example runs terribly slow. This increase in execution time is caused by the overheads needed to create a future for each data point. Because the work done within each call to heat is very small, the overhead of creating and scheduling each of the three futures is greater than that of the actual useful work! In order to amortize the overheads of our synchronization techniques, we need to be able to control the amount of work that will be done with each future. We call this amount of work per overhead grain size.

In example 3, we return to our serial code to figure out how to control the grain size of our program. The strategy that we employ is to create "partitions" of data points. The user can define how many partitions are created and how many data points are contained in each partition. This is accomplished by creating the `struct partition`, which contains a member object `data_`, a vector of doubles that holds the data points assigned to a particular instance of `partition`.

In example 4, we take advantage of the partition setup by redefining `space` to be a vector of shared_futures with each future representing a partition. In this manner, each future represents several data points. Because the user can define how many data points are in each partition, and, therefore, how many data points are represented by one future, a user can control the grainsize of the simulation. The rest of the code is then futurized in the same manner as example 2. It should be noted how strikingly similar example 4 is to example 2.

Example 4 finally shows good results. This code scales equivalently to the OpenMP version. While these results are promising, there are more opportunities to improve the application's scalability. Currently, this code only runs on one *locality*, but to get the full benefit of *HPX*, we need to be able to distribute the work to other machines in a cluster. We begin to add this functionality in example 5.

In order to run on a distributed system, a large amount of boilerplate code must be added. Fortunately, *HPX* provides us with the concept of a *component*, which saves us from having to write quite as much code. A component is an object that can be remotely accessed using its global address. Components are made of two parts: a server and a client class. While the client class is not required, abstracting the server behind a client allows us to ensure type safety instead of having to pass around pointers to global objects. Example 5 renames example 4's `struct partition` to `partition_data` and adds serialization support. Next, we add the server side representation of the data in the structure `partition_server`. `Partition_server` inherits from `hpx::components::component_base`, which contains a server-side component boilerplate. The boilerplate code allows a component's public members to be accessible anywhere on the machine via its Global Identifier (GID). To encapsulate the component, we create a client side helper class. This object allows us to create new instances of our component and access its members without having to know its GID. In addition, we are using the client class to assist us with managing our asynchrony. For example, our client class `partition`'s member function `get_data()` returns a future to `partition_data get_data()`. This struct inherits its boilerplate code from `hpx::components::client_base`.

In the structure `stepper`, we have also had to make some changes to accommodate a distributed environment. In order to get the data from a particular neighboring partition, which could be remote, we must retrieve the data from all of the neighboring partitions. These retrievals are asynchronous and the function `heat_part_data`, which, amongst other things, calls heat, should not be called unless the data from the neighboring partitions have arrived. Therefore, it should come as no surprise that we synchronize this operation with another instance of dataflow (found in heat_part). This dataflow receives futures to the data in the current and surrounding partitions by calling get_data() on each respective partition. When these futures are ready, dataflow passes them to the unwrapping function, which extracts the shared_array of doubles and passes them to the lambda. The lambda calls heat_part_data on the *locality*, which the middle partition is on.

Although this example could run distributed, it only runs on one *locality*, as it always uses hpx::find_here() as the target for the functions to run on.

In example 6, we begin to distribute the partition data on different nodes. This is accomplished in stepper::do_work() by passing the GID of the *locality* where we wish to create the partition to the partition constructor.

```
for (std::size_t i = 0; i != np; ++i)
    U[0][i] = partition(localities[locidx(i, np, nl)], nx, double(i));
```

We distribute the partitions evenly based on the number of localities used, which is described in the function locidx. Because some of the data needed to update the partition in heat_part could now be on a new *locality*, we must devise a way of moving data to the *locality* of the middle partition. We accomplished this by adding a switch in the function get_data() that returns the end element of the buffer data_ if it is from the left partition or the first element of the buffer if the data is from the right partition. In this way only the necessary elements, not the whole buffer, are exchanged between nodes. The reader should be reminded that this exchange of end elements occurs in the function get_data() and, therefore, is executed asynchronously.

Now that we have the code running in distributed, it is time to make some optimizations. The function heat_part spends most of its time on two tasks: retrieving remote data and working on the data in the middle partition. Because we know that the data for the middle partition is local, we can overlap the work on the middle partition with that of the possibly remote call of get_data(). This algorithmic change, which was implemented in example 7, can be seen below:

```
// The partitioned operator, it invokes the heat operator above on all elements
// of a partition.
static partition heat_part(
    partition const& left, partition const& middle, partition const& right)
{
    using hpx::dataflow;
    using hpx::unwrapping;

    hpx::shared_future<partition_data> middle_data =
        middle.get_data(partition_server::middle_partition);

    hpx::future<partition_data> next_middle = middle_data.then(
        unwrapping([middle](partition_data const& m) -> partition_data {
            HPX_UNUSED(middle);

            // All local operations are performed once the middle data of
            // the previous time step becomes available.
            std::size_t size = m.size();
            partition_data next(size);
            for (std::size_t i = 1; i != size - 1; ++i)
                next[i] = heat(m[i - 1], m[i], m[i + 1]);
            return next;
        }));
```

(continues on next page)

(continued from previous page)

```
        return dataflow(hpx::launch::async,
            unwrapping([left, middle, right](partition_data next,
                           partition_data const& l, partition_data const& m,
                           partition_data const& r) -> partition {
                HPX_UNUSED(left);
                HPX_UNUSED(right);

                // Calculate the missing boundary elements once the
                // corresponding data has become available.
                std::size_t size = m.size();
                next[0] = heat(l[size - 1], m[0], m[1]);
                next[size - 1] = heat(m[size - 2], m[size - 1], r[0]);

                // The new partition_data will be allocated on the same locality
                // as 'middle'.
                return partition(middle.get_id(), std::move(next));
            }),
            std::move(next_middle),
            left.get_data(partition_server::left_partition), middle_data,
            right.get_data(partition_server::right_partition));
    }
```

Example 8 completes the futurization process and utilizes the full potential of *HPX* by distributing the program flow to multiple localities, usually defined as nodes in a cluster. It accomplishes this task by running an instance of *HPX* main on each *locality*. In order to coordinate the execution of the program, the `struct stepper` is wrapped into a component. In this way, each *locality* contains an instance of stepper that executes its own instance of the function `do_work()`. This scheme does create an interesting synchronization problem that must be solved. When the program flow was being coordinated on the head node, the GID of each component was known. However, when we distribute the program flow, each partition has no notion of the GID of its neighbor if the next partition is on another *locality*. In order to make the GIDs of neighboring partitions visible to each other, we created two buffers to store the GIDs of the remote neighboring partitions on the left and right respectively. These buffers are filled by sending the GID of newly created edge partitions to the right and left buffers of the neighboring localities.

In order to finish the simulation, the solution vectors named `result` are then gathered together on *locality* 0 and added into a vector of spaces `overall_result` using the *HPX* functions `gather_id` and `gather_here`.

Example 8 completes this example series, which takes the serial code of example 1 and incrementally morphs it into a fully distributed parallel code. This evolution was guided by the simple principles of futurization, the knowledge of grainsize, and utilization of components. Applying these techniques easily facilitates the scalable parallelization of most applications.

2.2.8 Serializing user-defined types

In order to facilitate the sending and receiving of complex datatypes HPX provides a serialization abstraction.

Just like boost, hpx allows users to serialize user-defined types by either providing the serializer as a member function or defining the serialization as a free function.

Unlike Boost HPX doesn't acknowledge second unsigned int parameter, it is solely there to preserve API compatibility with Boost Serialization

This is tutorial was heavily inspired by Boost's serialization concepts[10].

[10] https://www.boost.org/doc/libs/1_79_0/libs/serialization/doc/serialization.html

Setup

The source code for this example can be found here: `custom_serialization.cpp`.

To compile this program, go to your *HPX* build directory (see *Building HPX* for information on configuring and building *HPX*) and enter:

```
$ make examples.quickstart.custom_serialization
```

To run the program type:

```
$ ./bin/custom_serialization
```

This should print:

```
Rectangle(Point(x=0,y=0),Point(x=0,y=5))
gravity.g = 9.81%
```

Serialization Requirements

In order to serialize objects in HPX, at least one of the following criteria must be met:

In the case of default constructible objects:

- The object is an empty type.
- Has a serialization function as shown in this tutorial.
- All members are accessible publicly and they can be used in structured binding contexts.

Otherwise:

- They need to have special serialization support.

Member function serialization

```
struct point_member_serialization
{
    int x{0};
    int y{0};

    // Required when defining the serialization function as private
    // In this case it isn't
    // Provides serialization access to HPX
    friend class hpx::serialization::access;

    // Second argument exists solely for compatibility with boost serialize
    // it is NOT processed by HPX in any way.
    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        // clang-format off
        ar & x & y;
        // clang-format on
    }
```

(continues on next page)

(continued from previous page)

```
};

// Allow bitwise serialization
HPX_IS_BITWISE_SERIALIZABLE(point_member_serialization)
```

Notice that `point_member_serialization` is defined as bitwise serializable (see *Bitwise serialization for bitwise copyable data* for more details). HPX is also able to recursively serialize composite classes and structs given that its members are serializable.

```
struct rectangle_member_serialization
{
    point_member_serialization top_left;
    point_member_serialization lower_right;

    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        // clang-format off
        ar & top_left & lower_right;
        // clang-format on
    }
};
```

Free function serialization

In order to decouple your models from HPX, HPX also allows for the definition of free function serializers.

```
struct rectangle_free
{
    point_member_serialization top_left;
    point_member_serialization lower_right;
};

template <typename Archive>
void serialize(Archive& ar, rectangle_free& pt, const unsigned int)
{
    // clang-format off
    ar & pt.lower_right & pt.top_left;
    // clang-format on
}
```

Even if you can't modify a class to befriend it, you can still be able to serialize your class provided that your class is default constructable and you are able to reconstruct it yourself.

```
class point_class
{
public:
    point_class(int x, int y)
      : x(x)
      , y(y)
    {
```

(continues on next page)

(continued from previous page)

```
    }

    point_class() = default;

    [[nodiscard]] int get_x() const noexcept
    {
        return x;
    }

    [[nodiscard]] int get_y() const noexcept
    {
        return y;
    }

private:
    int x;
    int y;
};

template <typename Archive>
void load(Archive& ar, point_class& pt, const unsigned int)
{
    int x, y;
    ar >> x >> y;
    pt = point_class(x, y);
}

template <typename Archive>
void save(Archive& ar, point_class const& pt, const unsigned int)
{
    ar << pt.get_x() << pt.get_y();
}

// This tells HPX that you have spilt your serialize function into
// load and save
HPX_SERIALIZATION_SPLIT_FREE(point_class)
```

Serializing non default constructable classes

Some classes don't provide any default constructor.

```
class planet_weight_calculator
{
public:
    explicit planet_weight_calculator(double g)
      : g(g)
    {
    }

    template <class Archive>
    friend void save_construct_data(
```

(continues on next page)

(continued from previous page)

```
        Archive&, planet_weight_calculator const*, unsigned int);

    [[nodiscard]] double get_g() const
    {
        return g;
    }

private:
    // Provides serialization access to HPX
    friend class hpx::serialization::access;
    template <class Archive>
    void serialize(Archive&, const unsigned int)
    {
        // Serialization will be done in the save_construct_data
        // Still needs to be defined
    }

    double g;
};
```

In this case you have to define a `save_construct_data` and `load_construct_data` in which you do the serialization yourself.

```
template <class Archive>
inline void save_construct_data(Archive& ar,
    planet_weight_calculator const* weight_calc, const unsigned int)
{
    ar << weight_calc->g;    // Do all of your serialization here
}

template <class Archive>
inline void load_construct_data(
    Archive& ar, planet_weight_calculator* weight_calc, const unsigned int)
{
    double g;
    ar >> g;

    // ::new(ptr) construct new object at given address
    hpx::construct_at(weight_calc, g);
}
```

Bitwise serialization for bitwise copyable data

When sending non arithmetic types not defined by std::is_arithmetic[11], HPX has to (de)serialize each object separately. However, if the class you are trying to send classes consists only of bitwise copyable datatypes, you may mark your class as such. Then HPX will serialize your object bitwise instead of element wise. This has enormous benefits, especially when sending a vector/array of your class. To define your class as such you need to call `HPX_IS_BITWISE_SERIALIZABLE(T)` with your desired custom class.

[11] https://en.cppreference.com/w/cpp/types/is_arithmetic

```
struct point_member_serialization
{
    int x{0};
    int y{0};

    // Required when defining the serialization function as private
    // In this case it isn't
    // Provides serialization access to HPX
    friend class hpx::serialization::access;

    // Second argument exists solely for compatibility with boost serialize
    // it is NOT processed by HPX in any way.
    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        // clang-format off
        ar & x & y;
        // clang-format on
    }
};

// Allow bitwise serialization
HPX_IS_BITWISE_SERIALIZABLE(point_member_serialization)
```

2.3 Manual

The manual is your comprehensive guide to *HPX*. It contains detailed information on how to build and use *HPX* in different scenarios.

2.3.1 Prerequisites

Supported platforms

At this time, *HPX* supports the following platforms. Other platforms may work, but we do not test *HPX* with other platforms, so please be warned.

Table 2.1: Supported Platforms for *HPX*

Name	Minimum Version	Architectures
Linux	2.6	x86-32, x86-64, k1om
BlueGeneQ	V1R2M0	PowerPC A2
Windows	Any Windows system	x86-32, x86-64
Mac OSX	Any OSX system	x86-64
ARM	Any ARM system	Any architecture
RISC-V	Any RISC-V system	Any architecture

Supported compilers

The table below shows the supported compilers for *HPX*.

Table 2.2: Supported Compilers for *HPX*

Name	Minimum Version
GNU Compiler Collection (g++)[12]	9.0
clang: a C language family frontend for LLVM[13]	10.0
Visual C++[14] (x64)	2019

Software and libraries

The table below presents all the necessary prerequisites for building *HPX*.

Table 2.3: Software prerequisites for *HPX*

	Name	Minimum Version
Build System	CMake[15]	3.18
Required Libraries	Boost[16]	1.71.0
	Portable Hardware Locality (HWLOC)[17]	1.5

The most important dependencies are Boost[18] and Portable Hardware Locality (HWLOC)[19]. The installation of Boost is described in detail in Boost's Getting Started[20] document. A recent version of hwloc is required in order to support thread pinning and NUMA awareness and can be found in Hwloc Downloads[21].

HPX is written in 99.99% Standard C++ (the remaining 0.01% is platform specific assembly code). As such, *HPX* is compilable with almost any standards compliant C++ compiler. The code base takes advantage of C++ language and standard library features when available.

Note: When building Boost using gcc, please note that it is required to specify a `cxxflags=-std=c++17` command line argument to `b2` (`bjam`).

Note: In most configurations, *HPX* depends only on header-only Boost. Boost.Filesystem is required if the standard library does not support filesystem. The following are not needed by default, but are required in certain configurations: Boost.Chrono, Boost.DateTime, Boost.Log, Boost.LogSetup, Boost.Regex, and Boost.Thread.

Depending on the options you chose while building and installing *HPX*, you will find that *HPX* may depend on several other libraries such as those listed below.

[12] https://gcc.gnu.org
[13] https://clang.llvm.org/
[14] https://msdn.microsoft.com/en-us/visualc/default.aspx
[15] https://www.cmake.org
[16] https://www.boost.org/
[17] https://www.open-mpi.org/projects/hwloc/
[18] https://www.boost.org/
[19] https://www.open-mpi.org/projects/hwloc/
[20] https://www.boost.org/more/getting_started/index.html
[21] https://www.open-mpi.org/software/hwloc/v1.11

Note: In order to use a high speed parcelport, we currently recommend configuring *HPX* to use MPI so that MPI can be used for communication between different localities. Please set the CMake variable `MPI_CXX_COMPILER` to your MPI C++ compiler wrapper if not detected automatically.

Table 2.4: Optional software prerequisites for *HPX*

Name	Minimum version
google-perftools[22]	1.7.1
jemalloc[23]	2.1.0
mi-malloc[24]	1.0.0
Performance Application Programming Interface (PAPI)	

2.3.2 Getting *HPX*

Download a tarball of the latest release from *HPX* Downloads[25] and unpack it or clone the repository directly using `git`:

```
$ git clone https://github.com/STEllAR-GROUP/hpx.git
```

It is also recommended that you check out the latest stable tag:

```
$ cd hpx
```

```
$ git checkout v1.10.0
```

2.3.3 Building *HPX*

Basic information

The build system for *HPX* is based on CMake[26], a cross-platform build-generator tool which is not responsible for building the project but rather generates the files needed by your build tool (GNU make, Visual Studio, etc.) for building *HPX*. If CMake is not already installed in your system, you can download it and install it here: CMake Downloads[27].

Once CMake[28] has been run, the build process can be started. The build process consists of the following parts:

- The *HPX* core libraries (target `core`): This forms the basic set of *HPX* libraries.
- *HPX* Examples (target `examples`): This target is enabled by default and builds all *HPX* examples (disable by setting *`HPX_WITH_EXAMPLES:BOOL`*`=Off`). *HPX* examples are part of the `all` target and are included in the installation if enabled.
- *HPX* Tests (target `tests`): This target builds the *HPX* test suite and is enabled by default (disable by setting *`HPX_WITH_TESTS:BOOL`* `=Off`). They are not built by the `all` target and have to be built separately.
- *HPX* Documentation (target `docs`): This target builds the documentation, and is not enabled by default (enable by setting *`HPX_WITH_DOCUMENTATION:BOOL`*`=On`. For more information see *Documentation*.

[22] https://code.google.com/p/gperftools
[23] http://jemalloc.net
[24] http://microsoft.github.io/mimalloc/
[25] https://hpx.stellar-group.org/downloads/
[26] https://www.cmake.org
[27] https://www.cmake.org/cmake/resources/software.html
[28] https://www.cmake.org

The *HPX* build process is highly configurable through CMake[29], and various CMake[30] variables influence the build process. A list with the most important CMake[31] variables can be found in the section that follows, while the complete list of available CMake[32] variables is in *CMake options*. These variables can be used to refine the recipes that can be found at *Platform specific build recipes*, a section that shows some basic steps on how to build *HPX* for a specific platform.

In order to use *HPX*, only the core libraries are required. In order to use the optional libraries, you need to specify them as link dependencies in your build (See *Creating HPX projects*).

Most important CMake options

While building *HPX*, you are provided with multiple CMake options which correspond to different configurations. Below, there is a set of the most important and frequently used CMake options.

`HPX_WITH_MALLOC`

Use a custom allocator. Using a custom allocator tuned for multithreaded applications is very important for the performance of *HPX* applications. When debugging applications, it's useful to set this to `system`, as custom allocators can hide some memory-related bugs. Note that setting this to something other than `system` requires an external dependency.

`HPX_WITH_CUDA`

Enable support for CUDA. Use `CMAKE_CUDA_COMPILER` to set the CUDA compiler. This is a standard CMake[33] variable, like `CMAKE_CXX_COMPILER`.

`HPX_WITH_PARCELPORT_MPI`

Enable the MPI parcelport. This enables the use of MPI for the networking operations in the HPX runtime. The default value is `OFF` because it's not available on all systems and/or requires another dependency. However, it is the recommended parcelport.

`HPX_WITH_PARCELPORT_TCP`

Enable the TCP parcelport. Enables the use of TCP for networking in the runtime. The default value is `ON`. However, it's only recommended for debugging purposes, as it is slower than the MPI parcelport.

`HPX_WITH_PARCELPORT_LCI`

Enable the LCI parcelport. This enables the use of LCI for the networking operations in the HPX runtime. The default value is `OFF` because it's not available on all systems and/or requires another dependency. However, this experimental parcelport may provide better performance than the MPI parcelport. Please refer to *Using the LCI parcelport* for more information about the LCI parcelport.

`HPX_WITH_APEX`

Enable APEX integration. APEX[34] can be used to profile *HPX* applications. In particular, it provides information about individual tasks in the *HPX* runtime.

`HPX_WITH_GENERIC_CONTEXT_COROUTINES`

Enable Boost. Context for task context switching. It must be enabled for non-x86 architectures such as ARM and Power.

`HPX_WITH_MAX_CPU_COUNT`

Set the maximum CPU count supported by *HPX*. The default value is 64, and should be set to a number at least as high as the number of cores on a system including virtual cores such as hyperthreads.

[29] https://www.cmake.org
[30] https://www.cmake.org
[31] https://www.cmake.org
[32] https://www.cmake.org
[33] https://www.cmake.org
[34] https://uo-oaciss.github.io/apex/quickstarthpx/

HPX_WITH_CXX_STANDARD

Set a specific C++ standard version e.g. `HPX_WITH_CXX_STANDARD=20`. The default and minimum value is 17.

HPX_WITH_EXAMPLES

Build examples.

HPX_WITH_TESTS

Build tests.

For a complete list of available CMake[35] variables that influence the build of *HPX*, see *CMake options*.

Build types

CMake[36] can be configured to generate project files suitable for builds that have enabled debugging support or for an optimized build (without debugging support). The CMake[37] variable used to set the build type is `CMAKE_BUILD_TYPE` (for more information see the CMake Documentation[38]). Available build types are:

- **Debug**: Full debug symbols are available as well as additional assertions to help debugging. To enable the debug build type for the *HPX* API, the C++ Macro `HPX_DEBUG` is defined.
- **RelWithDebInfo**: Release build with debugging symbols. This is most useful for profiling applications
- **Release**: Release build. This disables assertions and enables default compiler optimizations.
- **RelMinSize**: Release build with optimizations for small binary sizes.

Important: We currently don't guarantee ABI compatibility between Debug and Release builds. Please make sure that applications built against *HPX* use the same build type as you used to build *HPX*. For CMake builds, this means that the `CMAKE_BUILD_TYPE` variables have to match and for projects not using CMake[39], the `HPX_DEBUG` macro has to be set in debug mode.

Platform specific build recipes

Unix variants

Once you have the source code and the dependencies and assuming all your dependencies are in paths known to CMake[40], the following gets you started:

1. First, set up a separate build directory to configure the project:

```
$ mkdir build && cd build
```

2. To configure the project you have the following options:

   - To build the core *HPX* libraries and examples, and install them to your chosen location (recommended):

   ```
   $ cmake -DCMAKE_INSTALL_PREFIX=/install/path ..
   ```

[35] https://www.cmake.org
[36] https://www.cmake.org
[37] https://www.cmake.org
[38] https://cmake.org/cmake/help/latest/variable/CMAKE_BUILD_TYPE.html
[39] https://www.cmake.org
[40] https://www.cmake.org

Tip: If you want to change CMake[41] variables for your build, it is usually a good idea to start with a clean build directory to avoid configuration problems. It is especially important that you use a clean build directory when changing between `Release` and `Debug` modes.

- To install *HPX* to the default system folders, simply leave out the `CMAKE_INSTALL_PREFIX` option:

```
$ cmake ..
```

- If your dependencies are in custom locations, you may need to tell CMake[42] where to find them by passing one or more options to CMake[43] as shown below:

```
$ cmake -DBoost_ROOT=/path/to/boost
      -DHwloc_ROOT=/path/to/hwloc
      -DTcmalloc_ROOT=/path/to/tcmalloc
      -DJemalloc_ROOT=/path/to/jemalloc
      [other CMake variable definitions]
      /path/to/source/tree
```

For instance:

```
$ cmake -DBoost_ROOT=~/packages/boost -DHwloc_ROOT=/packages/hwloc -DCMAKE_
↪INSTALL_PREFIX=~/packages/hpx ~/downloads/hpx_1.5.1
```

- If you want to try *HPX* without using a custom allocator pass `-DHPX_WITH_MALLOC=system` to CMake[44]:

```
$ cmake -DCMAKE_INSTALL_PREFIX=/install/path -DHPX_WITH_MALLOC=system ..
```

Note: Please pay special attention to the section about *`HPX_WITH_MALLOC:STRING`* as this is crucial for getting decent performance.

Important: If you are building *HPX* for a system with more than 64 processing units, you must change the CMake[45] variable `HPX_WITH_MAX_CPU_COUNT` (to a value at least as big as the number of (virtual) cores on your system). Note that the default value is 64.

Caution: Compiling and linking *HPX* needs a considerable amount of memory. It is advisable that at least 2 GB of memory per parallel process is available.

3. Once the configuration is complete, to build the project you run:

```
$ cmake --build . --target install
```

41 https://www.cmake.org
42 https://www.cmake.org
43 https://www.cmake.org
44 https://www.cmake.org
45 https://www.cmake.org

Windows

Note: The following build recipes are mostly user-contributed and may be outdated. We always welcome updated and new build recipes.

To build *HPX* under Windows 10 x64 with Visual Studio 2015:

- Download the CMake V3.18.1 installer (or latest version) from here[46]
- Download the hwloc V1.11.0 (or the latest version) from here[47] and unpack it.
- Download the latest Boost libraries from here[48] and unpack them.
- Build the Boost DLLs and LIBs by using these commands from Command Line (or PowerShell). Open CMD/PowerShell inside the Boost dir and type in:

  ```
  .\bootstrap.bat
  ```

  This batch file will set up everything needed to create a successful build. Now execute:

  ```
  .\b2.exe link=shared variant=release,debug architecture=x86 address-model=64␣
  ↪threading=multi --build-type=complete install
  ```

  This command will start a (very long) build of all available Boost libraries. Please, be patient.

- Open CMake-GUI.exe and set up your source directory (input field 'Where is the source code') to the *base directory* of the source code you downloaded from *HPX*'s GitHub pages. Here's an example of CMake path settings, which point to the `Documents/GitHub/hpx` folder:

  Inside 'Where is the source-code' enter the base directory of your *HPX* source directory (do not enter the "src" sub-directory!). Inside 'Where to build the binaries' you should put in the path where all the building processes will happen. This is important because the building machinery will do an "out-of-tree" build. CMake will not touch or change the original source files in any way. Instead, it will generate Visual Studio Solution Files, which will build *HPX* packages out of the *HPX* source tree.

- Set new configuration variables (in CMake, not in Windows environment): `Boost_ROOT`, `Hwloc_ROOT`, `Asio_ROOT`, `CMAKE_INSTALL_PREFIX`. The meaning of these variables is as follows:
  - `Boost_ROOT` the *HPX* root directory of the unpacked Boost headers/cpp files.
  - `Hwloc_ROOT` the *HPX* root directory of the unpacked Portable Hardware Locality files.
  - `Asio_ROOT` the *HPX* root directory of the unpacked ASIO files. Alternatively use `HPX_WITH_FETCH_ASIO` with value `True`.
  - `CMAKE_INSTALL_PREFIX` the *HPX* root directory where the future builds of *HPX* should be installed.

  Note: *HPX* is a very large software collection, so it is not recommended to use the default `C:\Program Files\hpx`. Many users may prefer to use simpler paths *without* whitespace, like `C:\bin\hpx` or `D:\bin\hpx` etc.

  To insert new env-vars click on "Add Entry" and then insert the name inside "Name", select `PATH` as Type and put the path-name in the "Path" text field. Repeat this for the first three variables.

  This is how variable insertion will look:

[46] https://blog.kitware.com/cmake-3-18-1-available-for-download/
[47] http://www.open-mpi.org/software/hwloc/v1.11/downloads/hwloc-win64-build-1.11.0.zip
[48] https://www.boost.org/users/download/

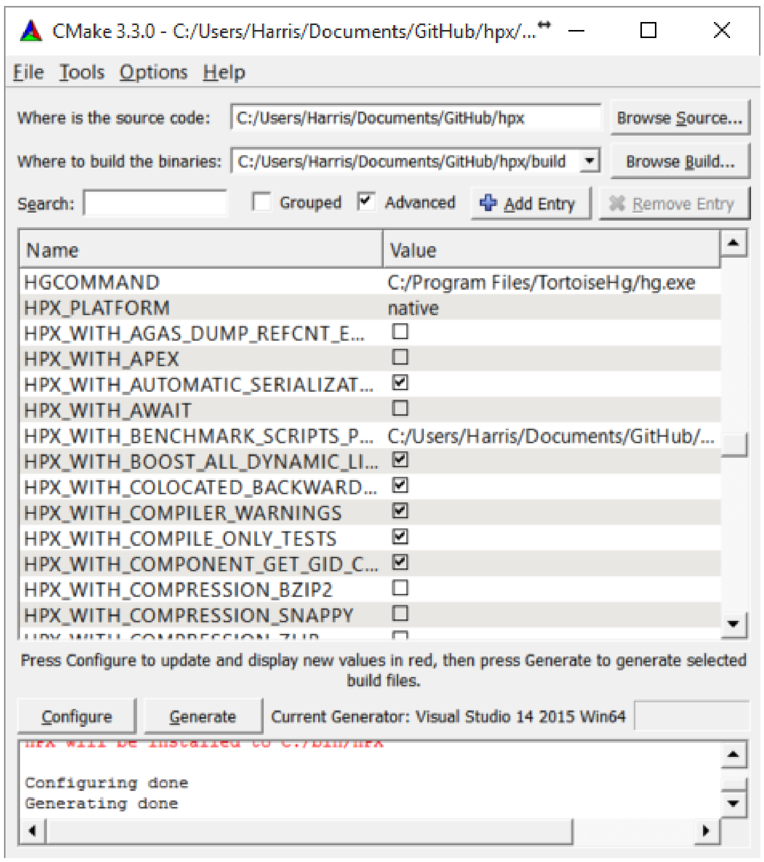

Fig. 2.3: Example CMake path settings.

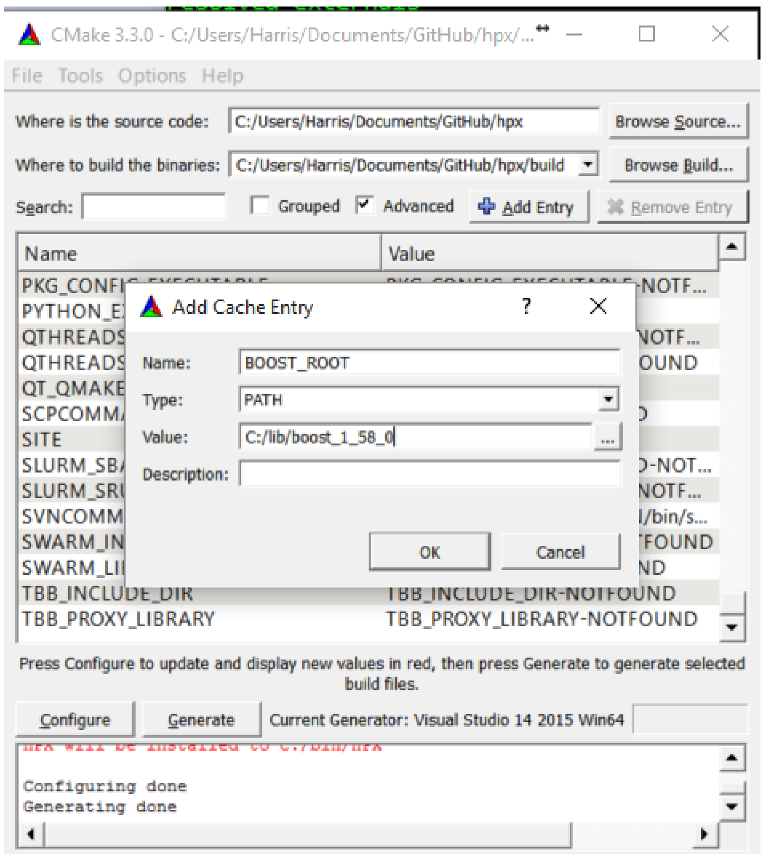

Fig. 2.4: Example CMake adding entry.

Alternatively, users could provide `Boost_LIBRARYDIR` instead of `Boost_ROOT`; the difference is that `Boost_LIBRARYDIR` should point to the subdirectory inside Boost root where all the compiled DLLs/LIBs are. For example, `Boost_LIBRARYDIR` may point to the `bin.v2` subdirectory under the Boost rootdir. It is important to keep the meanings of these two variables separated from each other: `Boost_DIR` points to the ROOT folder of the Boost library. `Boost_LIBRARYDIR` points to the subdir inside the Boost root folder where the compiled binaries are.

- Click the 'Configure' button of CMake-GUI. You will be immediately presented with a small window where you can select the C++ compiler to be used within Visual Studio. This has been tested using the latest v14 (a.k.a C++ 2015) but older versions should be sufficient too. Make sure to select the 64Bit compiler.
- After the generate process has finished successfully, click the 'Generate' button. Now, CMake will put new VS Solution files into the BUILD folder you selected at the beginning.
- Open Visual Studio and load the `HPX.sln` from your build folder.
- Go to `CMakePredefinedTargets` and build the `INSTALL` project:

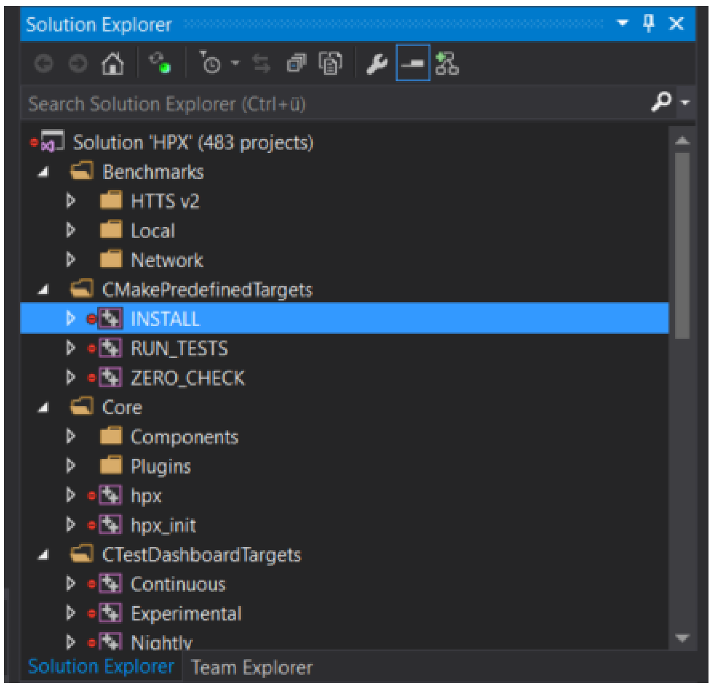

Fig. 2.5: Visual Studio INSTALL target.

It will take some time to compile everything, and in the end you should see an output similar to this one:

2.3.4 CMake options

In order to configure *HPX*, you can set a variety of options to allow CMake to generate your specific makefiles/project files. A list of the most important CMake options can be found in *Most important CMake options*, while this section includes the comprehensive list.

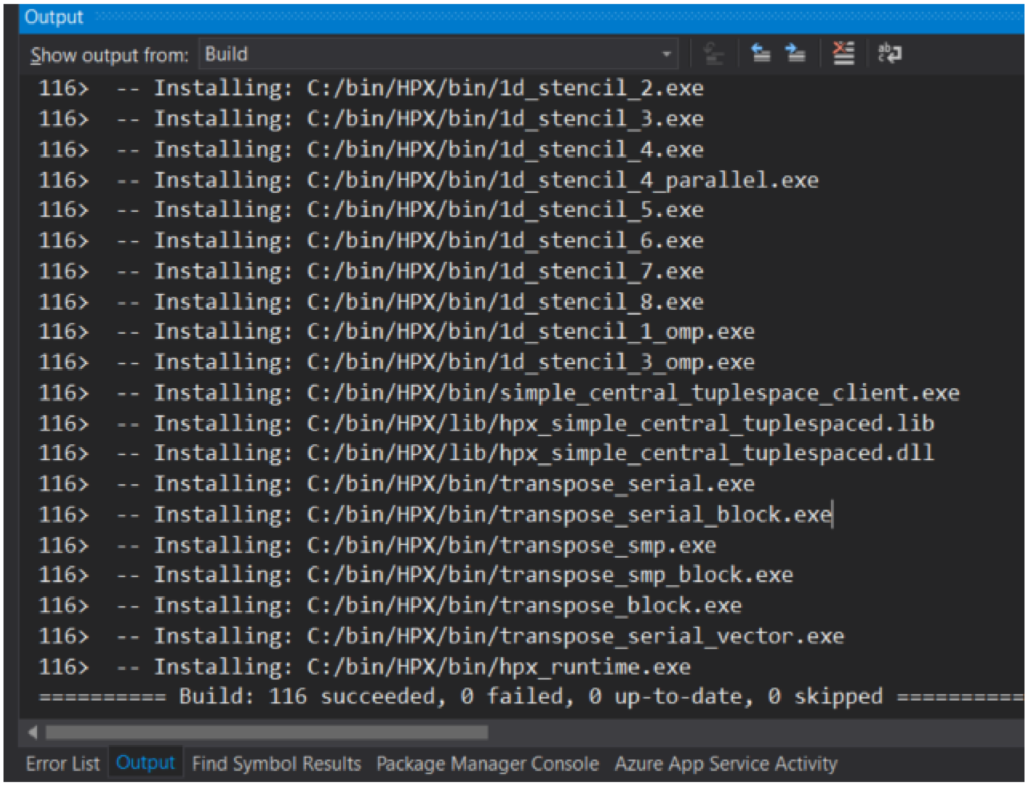

Fig. 2.6: Visual Studio build output.

Variables that influence how *HPX* is built

The options are split into these categories:

- *Generic options*
- *Build Targets options*
- *Thread Manager options*
- *AGAS options*
- *Parcelport options*
- *Profiling options*
- *Debugging options*
- *Modules options*

Generic options

- *HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL*
- *HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH*
- *HPX_WITH_BUILD_BINARY_PACKAGE:BOOL*
- *HPX_WITH_CHECK_MODULE_DEPENDENCIES:BOOL*
- *HPX_WITH_COMPILER_WARNINGS:BOOL*
- *HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL*
- *HPX_WITH_COMPRESSION_BZIP2:BOOL*
- *HPX_WITH_COMPRESSION_SNAPPY:BOOL*
- *HPX_WITH_COMPRESSION_ZLIB:BOOL*
- *HPX_WITH_CUDA:BOOL*
- *HPX_WITH_CXX_STANDARD:STRING*
- *HPX_WITH_DATAPAR:BOOL*
- *HPX_WITH_DATAPAR_BACKEND:STRING*
- *HPX_WITH_DATAPAR_VC_NO_LIBRARY:BOOL*
- *HPX_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL*
- *HPX_WITH_DYNAMIC_HPX_MAIN:BOOL*
- *HPX_WITH_FAULT_TOLERANCE:BOOL*
- *HPX_WITH_FULL_RPATH:BOOL*
- *HPX_WITH_GCC_VERSION_CHECK:BOOL*
- *HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL*
- *HPX_WITH_HIDDEN_VISIBILITY:BOOL*
- *HPX_WITH_HIP:BOOL*
- *HPX_WITH_HIPSYCL:BOOL*
- *HPX_WITH_IGNORE_COMPILER_COMPATIBILITY:BOOL*
- *HPX_WITH_LOGGING:BOOL*
- *HPX_WITH_MALLOC:STRING*
- *HPX_WITH_MODULES_AS_STATIC_LIBRARIES:BOOL*
- *HPX_WITH_NICE_THREADLEVEL:BOOL*
- *HPX_WITH_PARCEL_COALESCING:BOOL*
- *HPX_WITH_PKGCONFIG:BOOL*
- *HPX_WITH_PRECOMPILED_HEADERS:BOOL*
- *HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL*
- *HPX_WITH_STACKOVERFLOW_DETECTION:BOOL*
- *HPX_WITH_STATIC_LINKING:BOOL*

- *HPX_WITH_SUPPORT_NO_UNIQUE_ADDRESS_ATTRIBUTE:BOOL*
- *HPX_WITH_SYCL:BOOL*
- *HPX_WITH_SYCL_FLAGS:STRING*
- *HPX_WITH_UNITY_BUILD:BOOL*
- *HPX_WITH_VIM_YCM:BOOL*
- *HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING*

HPX_WITH_AUTOMATIC_SERIALIZATION_REGISTRATION:BOOL

Use automatic serialization registration for actions and functions. This affects compatibility between HPX applications compiled with different compilers (default ON)

HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH

Directory to place batch scripts in

HPX_WITH_BUILD_BINARY_PACKAGE:BOOL

Build HPX on the build infrastructure on any LINUX distribution (default: OFF).

HPX_WITH_CHECK_MODULE_DEPENDENCIES:BOOL

Verify that no modules are cross-referenced from a different module category (default: OFF)

HPX_WITH_COMPILER_WARNINGS:BOOL

Enable compiler warnings (default: ON)

HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL

Turn compiler warnings into errors (default: OFF)

HPX_WITH_COMPRESSION_BZIP2:BOOL

Enable bzip2 compression for parcel data (default: OFF).

HPX_WITH_COMPRESSION_SNAPPY:BOOL

Enable snappy compression for parcel data (default: OFF).

HPX_WITH_COMPRESSION_ZLIB:BOOL

Enable zlib compression for parcel data (default: OFF).

HPX_WITH_CUDA:BOOL

Enable support for CUDA (default: OFF)

HPX_WITH_CXX_STANDARD:STRING

Set the C++ standard to use when compiling HPX itself. (default: 17)

HPX_WITH_DATAPAR:BOOL

Enable data parallel algorithm support using Vc library (default: ON)

HPX_WITH_DATAPAR_BACKEND:STRING

Define which vectorization library should be used. Options are: VC, EVE, STD_EXPERIMENTAL_SIMD, SVE; NONE

HPX_WITH_DATAPAR_VC_NO_LIBRARY:BOOL

Don’t link with the Vc static library (default: OFF)

HPX_WITH_DEPRECATION_WARNINGS:BOOL

Enable warnings for deprecated facilities (default: ON).

HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL
Disables the mechanism that produces debug output for caught signals and unhandled exceptions (default: OFF)

HPX_WITH_DYNAMIC_HPX_MAIN:BOOL
Enable dynamic overload of system `main()` (Linux and Apple only, default: ON)

HPX_WITH_FAULT_TOLERANCE:BOOL
Build HPX to tolerate failures of nodes, i.e. ignore errors in active communication channels (default: OFF)

HPX_WITH_FULL_RPATH:BOOL
Build and link HPX libraries and executables with full RPATHs (default: ON)

HPX_WITH_GCC_VERSION_CHECK:BOOL
Don't ignore version reported by gcc (default: ON)

HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL
Use Boost.Context as the underlying coroutines context switch implementation.

HPX_WITH_HIDDEN_VISIBILITY:BOOL
Use -fvisibility=hidden for builds on platforms which support it (default OFF)

HPX_WITH_HIP:BOOL
Enable compilation with HIPCC (default: OFF)

HPX_WITH_HIPSYCL:BOOL
Use hipsycl cmake integration (default: OFF)

HPX_WITH_IGNORE_COMPILER_COMPATIBILITY:BOOL
Ignore compiler incompatibility in dependent projects (default: ON).

HPX_WITH_LOGGING:BOOL
Build HPX with logging enabled (default: ON).

HPX_WITH_MALLOC:STRING
Define which allocator should be linked in. Options are: system, tcmalloc, jemalloc, mimalloc, tbbmalloc, and custom (default is: tcmalloc)

HPX_WITH_MODULES_AS_STATIC_LIBRARIES:BOOL
Compile HPX modules as STATIC (whole-archive) libraries instead of OBJECT libraries (Default: ON)

HPX_WITH_NICE_THREADLEVEL:BOOL
Set HPX worker threads to have high NICE level (may impact performance) (default: OFF)

HPX_WITH_PARCEL_COALESCING:BOOL
Enable the parcel coalescing plugin (default: ON).

HPX_WITH_PKGCONFIG:BOOL
Enable generation of pkgconfig files (default: ON on Linux without CUDA/HIP, otherwise OFF)

HPX_WITH_PRECOMPILED_HEADERS:BOOL
Enable precompiled headers for certain build targets (experimental) (default OFF)

HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL
Run hpx_main by default on all localities (default: OFF).

HPX_WITH_STACKOVERFLOW_DETECTION:BOOL
Enable stackoverflow detection for HPX threads/coroutines (default: OFF, debug: ON).

HPX_WITH_STATIC_LINKING:BOOL
Compile HPX statically linked libraries (Default: OFF)

HPX_WITH_SUPPORT_NO_UNIQUE_ADDRESS_ATTRIBUTE:BOOL
Enable the use of the [[no_unique_address]] attribute (default: ON)

HPX_WITH_SYCL:BOOL
Enable support for Sycl (default: OFF)

HPX_WITH_SYCL_FLAGS:STRING
Sycl compile flags for selecting specific targets (default: empty)

HPX_WITH_UNITY_BUILD:BOOL
Enable unity build for certain build targets (default OFF)

HPX_WITH_VIM_YCM:BOOL
Generate HPX completion file for VIM YouCompleteMe plugin

HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING
The threshold in bytes to when perform zero copy optimizations (default: 8192)

Build Targets options

- *HPX_WITH_ASIO_TAG:STRING*
- *HPX_WITH_COMPILE_ONLY_TESTS:BOOL*
- *HPX_WITH_DISTRIBUTED_RUNTIME:BOOL*
- *HPX_WITH_DOCUMENTATION:BOOL*
- *HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS:STRING*
- *HPX_WITH_EXAMPLES:BOOL*
- *HPX_WITH_EXAMPLES_HDF5:BOOL*
- *HPX_WITH_EXAMPLES_OPENMP:BOOL*
- *HPX_WITH_EXAMPLES_QT4:BOOL*
- *HPX_WITH_EXAMPLES_QTHREADS:BOOL*
- *HPX_WITH_EXAMPLES_TBB:BOOL*
- *HPX_WITH_EXECUTABLE_PREFIX:STRING*
- *HPX_WITH_FAIL_COMPILE_TESTS:BOOL*
- *HPX_WITH_FETCH_APEX:BOOL*
- *HPX_WITH_FETCH_ASIO:BOOL*
- *HPX_WITH_FETCH_BOOST:BOOL*
- *HPX_WITH_FETCH_GASNET:BOOL*
- *HPX_WITH_FETCH_HWLOC:BOOL*
- *HPX_WITH_FETCH_LCI:BOOL*
- *HPX_WITH_IO_COUNTERS:BOOL*
- *HPX_WITH_LCI_TAG:STRING*

- *HPX_WITH_PARALLEL_LINK_JOBS:STRING*
- *HPX_WITH_TESTS:BOOL*
- *HPX_WITH_TESTS_BENCHMARKS:BOOL*
- *HPX_WITH_TESTS_EXAMPLES:BOOL*
- *HPX_WITH_TESTS_EXTERNAL_BUILD:BOOL*
- *HPX_WITH_TESTS_HEADERS:BOOL*
- *HPX_WITH_TESTS_REGRESSIONS:BOOL*
- *HPX_WITH_TESTS_UNIT:BOOL*
- *HPX_WITH_TOOLS:BOOL*

HPX_WITH_ASIO_TAG:STRING

Asio repository tag or branch

HPX_WITH_COMPILE_ONLY_TESTS:BOOL

Create build system support for compile time only HPX tests (default ON)

HPX_WITH_DISTRIBUTED_RUNTIME:BOOL

Enable the distributed runtime (default: ON). Turning off the distributed runtime completely disallows the creation and use of components and actions. Turning this option off is experimental!

HPX_WITH_DOCUMENTATION:BOOL

Build the HPX documentation (default OFF).

HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS:STRING

List of documentation output formats to generate. Valid options are html;singlehtml;latexpdf;man. Multiple values can be separated with semicolons. (default html).

HPX_WITH_EXAMPLES:BOOL

Build the HPX examples (default ON)

HPX_WITH_EXAMPLES_HDF5:BOOL

Enable examples requiring HDF5 support (default: OFF).

HPX_WITH_EXAMPLES_OPENMP:BOOL

Enable examples requiring OpenMP support (default: OFF).

HPX_WITH_EXAMPLES_QT4:BOOL

Enable examples requiring Qt4 support (default: OFF).

HPX_WITH_EXAMPLES_QTHREADS:BOOL

Enable examples requiring QThreads support (default: OFF).

HPX_WITH_EXAMPLES_TBB:BOOL

Enable examples requiring TBB support (default: OFF).

HPX_WITH_EXECUTABLE_PREFIX:STRING

Executable prefix (default none), '**hpx_**' useful for system install.

HPX_WITH_FAIL_COMPILE_TESTS:BOOL

Create build system support for fail compile HPX tests (default ON)

HPX_WITH_FETCH_APEX:BOOL

Use FetchContent to fetch APEX. By default an installed APEX will be used. (default: OFF)

HPX_WITH_FETCH_ASIO:BOOL
Use FetchContent to fetch Asio. By default an installed Asio will be used. (default: OFF)

HPX_WITH_FETCH_BOOST:BOOL
Use FetchContent to fetch Boost. By default an installed Boost will be used. (default: OFF)

HPX_WITH_FETCH_GASNET:BOOL
Use FetchContent to fetch GASNET. By default an installed GASNET will be used. (default: OFF).

HPX_WITH_FETCH_HWLOC:BOOL
Use FetchContent to fetch Hwloc. By default an installed Hwloc will be used. (default: OFF)

HPX_WITH_FETCH_LCI:BOOL
Use FetchContent to fetch LCI. By default an installed LCI will be used. (default: OFF)

HPX_WITH_IO_COUNTERS:BOOL
Enable IO counters (default: ON)

HPX_WITH_LCI_TAG:STRING
LCI repository tag or branch

HPX_WITH_PARALLEL_LINK_JOBS:STRING
Number of Parallel link jobs while building hpx (only for Ninja as generator) (default 2)

HPX_WITH_TESTS:BOOL
Build the HPX tests (default ON)

HPX_WITH_TESTS_BENCHMARKS:BOOL
Build HPX benchmark tests (default: ON)

HPX_WITH_TESTS_EXAMPLES:BOOL
Add HPX examples as tests (default: ON)

HPX_WITH_TESTS_EXTERNAL_BUILD:BOOL
Build external cmake build tests (default: ON)

HPX_WITH_TESTS_HEADERS:BOOL
Build HPX header tests (default: OFF)

HPX_WITH_TESTS_REGRESSIONS:BOOL
Build HPX regression tests (default: ON)

HPX_WITH_TESTS_UNIT:BOOL
Build HPX unit tests (default: ON)

HPX_WITH_TOOLS:BOOL
Build HPX tools (default: OFF)

Thread Manager options

- *HPX_COROUTINES_WITH_SWAP_CONTEXT_EMULATION:BOOL*
- *HPX_COROUTINES_WITH_THREAD_SCHEDULE_HINT_RUNS_AS_CHILD:BOOL*
- *HPX_WITH_COROUTINE_COUNTERS:BOOL*
- *HPX_WITH_IO_POOL:BOOL*
- *HPX_WITH_MAX_CPU_COUNT:STRING*

- *HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING*
- *HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL*
- *HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL*
- *HPX_WITH_SPINLOCK_POOL_NUM:STRING*
- *HPX_WITH_STACKTRACES:BOOL*
- *HPX_WITH_STACKTRACES_DEMANGLE_SYMBOLS:BOOL*
- *HPX_WITH_STACKTRACES_STATIC_SYMBOLS:BOOL*
- *HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING*
- *HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL*
- *HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL*
- *HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL*
- *HPX_WITH_THREAD_IDLE_RATES:BOOL*
- *HPX_WITH_THREAD_LOCAL_STORAGE:BOOL*
- *HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL*
- *HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL*
- *HPX_WITH_THREAD_STACK_MMAP:BOOL*
- *HPX_WITH_THREAD_STEALING_COUNTS:BOOL*
- *HPX_WITH_THREAD_TARGET_ADDRESS:BOOL*
- *HPX_WITH_TIMER_POOL:BOOL*
- *HPX_WITH_WORK_REQUESTING_SCHEDULERS:BOOL*

HPX_COROUTINES_WITH_SWAP_CONTEXT_EMULATION:BOOL

Emulate SwapContext API for coroutines (Windows only, default: OFF)

HPX_COROUTINES_WITH_THREAD_SCHEDULE_HINT_RUNS_AS_CHILD:BOOL

Futures attempt to run associated threads directly if those have not been started (default: OFF)

HPX_WITH_COROUTINE_COUNTERS:BOOL

Enable keeping track of coroutine creation and rebind counts (default: OFF)

HPX_WITH_IO_POOL:BOOL

Disable internal IO thread pool, do not change if not absolutely necessary (default: ON)

HPX_WITH_MAX_CPU_COUNT:STRING

HPX applications will not use more that this number of OS-Threads (empty string means dynamic) (default: “”)

HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING

HPX applications will not run on machines with more NUMA domains (default: 8)

HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL

Enable scheduler local storage for all HPX schedulers (default: OFF)

HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL

Enable spinlock deadlock detection (default: OFF)

`HPX_WITH_SPINLOCK_POOL_NUM:STRING`
: Number of elements a spinlock pool manages (default: 128)

`HPX_WITH_STACKTRACES:BOOL`
: Attach backtraces to HPX exceptions (default: ON)

`HPX_WITH_STACKTRACES_DEMANGLE_SYMBOLS:BOOL`
: Thread stack back trace symbols will be demangled (default: ON)

`HPX_WITH_STACKTRACES_STATIC_SYMBOLS:BOOL`
: Thread stack back trace will resolve static symbols (default: OFF)

`HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING`
: Thread stack back trace depth being captured (default: 20)

`HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL`
: Enable thread stack back trace being captured on suspension (default: OFF)

`HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL`
: Enable measuring thread creation and cleanup times (default: OFF)

`HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL`
: Enable keeping track of cumulative thread counts in the schedulers (default: ON)

`HPX_WITH_THREAD_IDLE_RATES:BOOL`
: Enable measuring the percentage of overhead times spent in the scheduler (default: OFF)

`HPX_WITH_THREAD_LOCAL_STORAGE:BOOL`
: Enable thread local storage for all HPX threads (default: OFF)

`HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL`
: HPX scheduler threads do exponential backoff on idle queues (default: ON)

`HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL`
: Enable collecting queue wait times for threads (default: OFF)

`HPX_WITH_THREAD_STACK_MMAP:BOOL`
: Use mmap for stack allocation on appropriate platforms

`HPX_WITH_THREAD_STEALING_COUNTS:BOOL`
: Enable keeping track of counts of thread stealing incidents in the schedulers (default: OFF)

`HPX_WITH_THREAD_TARGET_ADDRESS:BOOL`
: Enable storing target address in thread for NUMA awareness (default: OFF)

`HPX_WITH_TIMER_POOL:BOOL`
: Disable internal timer thread pool, do not change if not absolutely necessary (default: ON)

`HPX_WITH_WORK_REQUESTING_SCHEDULERS:BOOL`
: Enable work requesting scheduler (default: ON)

AGAS options

- *HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL*

HPX_WITH_AGAS_DUMP_REFCNT_ENTRIES:BOOL

Enable dumps of the AGAS refcnt tables to logs (default: OFF)

Parcelport options

- *HPX_WITH_NETWORKING:BOOL*
- *HPX_WITH_PARCELPORT_ACTION_COUNTERS:BOOL*
- *HPX_WITH_PARCELPORT_COUNTERS:BOOL*
- *HPX_WITH_PARCELPORT_GASNET:BOOL*
- *HPX_WITH_PARCELPORT_LCI:BOOL*
- *HPX_WITH_PARCELPORT_LCI_LOG:STRING*
- *HPX_WITH_PARCELPORT_LCI_PCOUNTER:STRING*
- *HPX_WITH_PARCELPORT_LIBFABRIC:BOOL*
- *HPX_WITH_PARCELPORT_MPI:BOOL*
- *HPX_WITH_PARCELPORT_TCP:BOOL*
- *HPX_WITH_PARCEL_PROFILING:BOOL*

HPX_WITH_NETWORKING:BOOL

Enable support for networking and multi-node runs (default: ON)

HPX_WITH_PARCELPORT_ACTION_COUNTERS:BOOL

Enable performance counters reporting parcelport statistics on a per-action basis.

HPX_WITH_PARCELPORT_COUNTERS:BOOL

Enable performance counters reporting parcelport statistics.

HPX_WITH_PARCELPORT_GASNET:BOOL

Enable the GASNET based parcelport.

HPX_WITH_PARCELPORT_LCI:BOOL

Enable the LCI based parcelport.

HPX_WITH_PARCELPORT_LCI_LOG:STRING

Enable the LCI-parcelport-specific logger

HPX_WITH_PARCELPORT_LCI_PCOUNTER:STRING

Enable the LCI-parcelport-specific performance counter

HPX_WITH_PARCELPORT_LIBFABRIC:BOOL

Enable the libfabric based parcelport. This is currently an experimental feature

HPX_WITH_PARCELPORT_MPI:BOOL

Enable the MPI based parcelport.

HPX_WITH_PARCELPORT_TCP:BOOL

Enable the TCP based parcelport.

HPX_WITH_PARCEL_PROFILING:BOOL

Enable profiling data for parcels

Profiling options

- *HPX_WITH_APEX:BOOL*
- *HPX_WITH_ITTNOTIFY:BOOL*
- *HPX_WITH_PAPI:BOOL*

HPX_WITH_APEX:BOOL

Enable APEX instrumentation support.

HPX_WITH_ITTNOTIFY:BOOL

Enable Amplifier (ITT) instrumentation support.

HPX_WITH_PAPI:BOOL

Enable the PAPI based performance counter.

Debugging options

- *HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL*
- *HPX_WITH_PARALLEL_TESTS_BIND_NONE:BOOL*
- *HPX_WITH_SANITIZERS:BOOL*
- *HPX_WITH_TESTS_COMMAND_LINE:STRING*
- *HPX_WITH_TESTS_DEBUG_LOG:BOOL*
- *HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING*
- *HPX_WITH_TESTS_MAX_THREADS_PER_LOCALITY:STRING*
- *HPX_WITH_THREAD_DEBUG_INFO:BOOL*
- *HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL*
- *HPX_WITH_THREAD_GUARD_PAGE:BOOL*
- *HPX_WITH_VALGRIND:BOOL*
- *HPX_WITH_VERIFY_LOCKS:BOOL*
- *HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL*

HPX_WITH_ATTACH_DEBUGGER_ON_TEST_FAILURE:BOOL

Break the debugger if a test has failed (default: OFF)

HPX_WITH_PARALLEL_TESTS_BIND_NONE:BOOL

Pass –hpx:bind=none to tests that may run in parallel (cmake -j flag) (default: OFF)

HPX_WITH_SANITIZERS:BOOL

Configure with sanitizer instrumentation support.

HPX_WITH_TESTS_COMMAND_LINE:STRING

Add given command line options to all tests run

HPX_WITH_TESTS_DEBUG_LOG:BOOL

Turn on debug logs (–hpx:debug-hpx-log) for tests (default: OFF)

HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING

Destination for test debug logs (default: cout)

HPX_WITH_TESTS_MAX_THREADS_PER_LOCALITY:STRING

Maximum number of threads to use for tests (default: 0, use the number of threads specified by the test)

HPX_WITH_THREAD_DEBUG_INFO:BOOL

Enable thread debugging information (default: OFF, implicitly enabled in debug builds)

HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL

Use function address for thread description (default: OFF)

HPX_WITH_THREAD_GUARD_PAGE:BOOL

Enable thread guard page (default: ON)

HPX_WITH_VALGRIND:BOOL

Enable Valgrind instrumentation support.

HPX_WITH_VERIFY_LOCKS:BOOL

Enable lock verification code (default: OFF, enabled in debug builds)

HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL

Enable thread stack back trace being captured on lock registration (to be used in combination with HPX_WITH_VERIFY_LOCKS=ON, default: OFF)

Modules options

- *HPX_ALLOCATOR_SUPPORT_WITH_CACHING:BOOL*
- *HPX_COMMAND_LINE_HANDLING_LOCAL_WITH_JSON_CONFIGURATION_FILES:BOOL*
- *HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL*
- *HPX_DATASTRUCTURES_WITH_ADAPT_STD_VARIANT:BOOL*
- *HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY:BOOL*
- *HPX_ITERATOR_SUPPORT_WITH_BOOST_ITERATOR_TRAVERSAL_TAG_COMPATIBILITY:BOOL*
- *HPX_LOGGING_WITH_SEPARATE_DESTINATIONS:BOOL*
- *HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS:BOOL*
- *HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION:BOOL*
- *HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE:BOOL*
- *HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL*
- *HPX_SERIALIZATION_WITH_SUPPORTS_ENDIANESS:BOOL*
- *HPX_TOPOLOGY_WITH_ADDITIONAL_HWLOC_TESTING:BOOL*
- *HPX_WITH_POWER_COUNTER:BOOL*

HPX_ALLOCATOR_SUPPORT_WITH_CACHING:BOOL

Enable caching allocator. (default: ON)

HPX_COMMAND_LINE_HANDLING_LOCAL_WITH_JSON_CONFIGURATION_FILES:BOOL

Enable reading JSON formatted configuration files on the command line.

(default: On)

HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL

Enable compatibility of hpx::get with std::tuple. (default: ON)

HPX_DATASTRUCTURES_WITH_ADAPT_STD_VARIANT:BOOL

Enable compatibility of hpx::get with std::variant.

(default: OFF)

HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY:BOOL

Enable Boost.FileSystem compatibility. (default: OFF)

HPX_ITERATOR_SUPPORT_WITH_BOOST_ITERATOR_TRAVERSAL_TAG_COMPATIBILITY:BOOL

Enable Boost.Iterator traversal tag compatibility. (default: OFF)

HPX_LOGGING_WITH_SEPARATE_DESTINATIONS:BOOL

Enable separate logging channels for AGAS, timing, and parcel transport. (default: ON)

HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS:BOOL

Enable serializing std::tuple with const members. (default: OFF)

HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION:BOOL

Enable serializing raw pointers. (default: OFF)

HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE:BOOL

Assume all types are bitwise serializable. (default: OFF)

HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL

Enable serialization of certain Boost types. (default: OFF)

HPX_SERIALIZATION_WITH_SUPPORTS_ENDIANESS:BOOL

Support endian conversion on inout and output archives. (default: OFF)

HPX_TOPOLOGY_WITH_ADDITIONAL_HWLOC_TESTING:BOOL

Enable HWLOC filtering that makes it report no cores, this is purely an

option supporting better testing - do not enable under normal circumstances. (default: OFF)

HPX_WITH_POWER_COUNTER:BOOL

Enable use of performance counters based on pwr library (default: OFF)

Additional tools and libraries used by *HPX*

Here is a list of additional libraries and tools that are either optionally supported by the build system or are optionally required for certain examples or tests. These libraries and tools can be detected by the *HPX* build system.

Each of the tools or libraries listed here will be automatically detected if they are installed in some standard location. If a tool or library is installed in a different location, you can specify its base directory by appending `_ROOT` to the variable name as listed below. For instance, to configure a custom directory for `Boost`, specify `Boost_ROOT=/custom/boost/root`.

Boost_ROOT:PATH

Specifies where to look for the Boost installation to be used for compiling *HPX*. Set this if CMake is not able to locate a suitable version of Boost. The directory specified here can be either the root of an installed Boost distribution or the directory where you unpacked and built Boost without installing it (with staged libraries).

Hwloc_ROOT:PATH

Specifies where to look for the hwloc library. Set this if CMake is not able to locate a suitable version of hwloc. Hwloc provides platform- independent support for extracting information about the used hardware architecture (number of cores, number of NUMA domains, hyperthreading, etc.). *HPX* utilizes this information if available.

Papi_ROOT:PATH

Specifies where to look for the PAPI library. The PAPI library is needed to compile a special component exposing PAPI hardware events and counters as *HPX* performance counters. This is not available on the Windows platform.

Amplifier_ROOT:PATH

Specifies where to look for one of the tools of the Intel Parallel Studio product, either Intel Amplifier or Intel Inspector. This should be set if the CMake variable `HPX_USE_ITT_NOTIFY` is set to `ON`. Enabling ITT support in *HPX* will integrate any application with the mentioned Intel tools, which customizes the generated information for your application and improves the generated diagnostics.

In addition, some of the examples may need the following variables:

Hdf5_ROOT:PATH

Specifies where to look for the Hierarchical Data Format V5 (HDF5) include files and libraries.

2.3.5 Migration guide

The Migration Guide serves as a valuable resource for developers seeking to transition their parallel computing applications from different APIs (i.e. OpenMP, Intel Threading Building Blocks (TBB), MPI) to *HPX*. *HPX*, an advanced C++ library, offers a versatile and high-performance platform for parallel and distributed computing, providing a wide range of features and capabilities. This guide aims to assist developers in understanding the key differences between different APIs and *HPX*, and it provides step-by-step instructions for converting code to *HPX* code effectively.

Some general steps that can be used to migrate code to *HPX* code are the following:

1. Install *HPX* using the *Quick start* guide.
2. Include the *HPX* header files:

   Add the necessary header files for HPX at the beginning of your code, such as:

   ```
   #include <hpx/init.hpp>
   ```

3. Replace your code with *HPX* code using the guide that follows.
4. Use HPX-specific features and APIs:

   HPX provides additional features and APIs that can be used to take advantage of the library's capabilities. For example, you can use the HPX asynchronous execution to express fine-grained tasks and dependencies, or utilize HPX's distributed computing features for distributed memory systems.

5. Compile and run the *HPX* code:

   Compile the converted code with the *HPX* library and run it using the appropriate HPX runtime environment.

OpenMP

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++. Typically it is used for loop-level parallelism, but it also supports function-level parallelism. Below are some examples on how to convert OpenMP to *HPX* code:

OpenMP parallel for loop

Parallel for loop

OpenMP code:

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    // loop body
}
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>

hpx::experimental::for_loop(hpx::execution::par, 0, n, [&](int i) {
    // loop body
});
```

In the above code, the OpenMP *#pragma omp parallel for* directive is replaced with `hpx::experimental::for_loop` from the *HPX* library. The loop body within the lambda function will be executed in parallel for each iteration.

Private variables

OpenMP code:

```
int x = 0;

#pragma omp parallel for private(x)
for (int i = 0; i < n; ++i) {
    // loop body
}
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>

hpx::experimental::for_loop(hpx::execution::par, 0, n, [&](int i) {
        int x = 0; // Declare 'x' as a local variable inside the loop body
        // loop body
});
```

The variable *x* is declared as a local variable inside the loop body, ensuring that it is private to each thread.

Shared variables

OpenMP code:

```
int x = 0;

#pragma omp parallel for shared(x)
for (int i = 0; i < n; ++i) {
    // loop body
}
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>

std::atomic<int> x = 0; // Declare 'x' as a shared variable outside the loop

hpx::experimental::for_loop(hpx::execution::par, 0, n, [&](int i) {
    // loop body
});
```

To ensure variable *x* is shared among all threads, you simply have to declare it as an atomic variable outside the *for_loop*.

Number of threads

OpenMP code:

```
#pragma omp parallel for num_threads(2)
for (int i = 0; i < n; ++i) {
    // loop body
}
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>
#include <hpx/execution.hpp>

hpx::execution::experimental::num_cores nc(2);

hpx::experimental::for_loop(hpx::execution::par.with(nc), 0, n, [&](int i) {
    // loop body
});
```

To declare the number of threads to be used for the parallel region, you can use *hpx::execution::experimental::num_cores* and pass the number of cores (*nc*) to `hpx::experimental::for_loop` using *hpx::execution::par.with(nc)*. This example uses 2 threads for the parallel loop.

Reduction

OpenMP code:

```
int s = 0;

#pragma omp parallel for reduction(+: s)
for (int i = 0; i < n; ++i) {
    s += i;
    // loop body
}
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>
#include <hpx/execution.hpp>

int s = 0;

hpx::experimental::for_loop(hpx::execution::par, 0, n, reduction(s, 0, plus<>()), [&
↪](int i, int& accum) {
    accum += i;
    // loop body
});
```

The reduction clause specifies that the variable *s*` should be reduced across iterations using the *plus<>*` operation. It initializes *s* to *0* at the beginning of the loop and accumulates the values of *s* from each iteration using the + operator. The lambda function representing the loop body takes two parameters: *i*, which represents the loop index, and *accum*, which is the reduction variable *s*. The lambda function is executed for each iteration of the loop. The reduction ensures that the *accum* value is correctly accumulated across different iterations and threads.

Schedule

OpenMP code:

```
int s = 0;

// static scheduling with chunk size 1000
#pragma omp parallel for schedule(static, 1000)
for (int i = 0; i < n; ++i) {
    // loop body
}
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>
#include <hpx/execution.hpp>

hpx::execution::experimental::static_chunk_size cs(1000);

hpx::experimental::for_loop(hpx::execution::par.with(cs), 0, n, [&](int i) {
    // loop body
});
```

To define the scheduling type, you can use the corresponding execution policy from *hpx::execution::experimental*, define the chunk size (cs, here declared as 1000) and pass it to the to `hpx::experimental::for_loop` using *hpx::execution::par.with(cs)*.

Accordingly, other types of scheduling are available and can be used in a similar manner:

```
#include <hpx/execution.hpp>
hpx::execution::experimental::dynamic_chunk_size cs(1000);
```

```
#include <hpx/execution.hpp>
hpx::execution::experimental::guided_chunk_size cs(1000);
```

```
#include <hpx/execution.hpp>
hpx::execution::experimental::auto_chunk_size cs(1000);
```

OpenMP single thread

OpenMP code:

```
{   // parallel code
    #pragma omp single
    {
        // single-threaded code
    }
    // more parallel code
}
```

HPX equivalent:

```
#include <hpx/mutex.hpp>

hpx::mutex mtx;

{   // parallel code
    {   // single-threaded code
        std::scoped_lock l(mtx);
    }
    // more parallel code
}
```

To make sure that only one thread accesses a specific code within a parallel section you can use `hpx::mutex` and *std::scoped_lock* to take ownership of the given mutex *mtx*. For more information about mutexes please refer to *Mutex*.

OpenMP tasks

Simple tasks

OpenMP code:

```
// executed asynchronously by any available thread
#pragma omp task
{
    // task code
}
```

HPX equivalent:

```
#include <hpx/future.hpp>

auto future = hpx::async([](){
    // task code
});
```

or

```
#include <hpx/future.hpp>

hpx::post([](){
    // task code
}); // fire and forget
```

The tasks in *HPX* can be defined simply by using the `async` function and passing as argument the code you wish to run asynchronously. Another alternative is to use `post` which is a fire-and-forget method.

Tip: If you think you will like to synchronize your tasks later on, we suggest you use `hpx::async` which provides synchronization options, while `hpx::post` explicitly states that there is no return value or way to synchronize with the function execution. Synchronization options are listed below.

Task wait

OpenMP code:

```
#pragma omp task
{
    // task code
}

#pragma omp taskwait
// code after completion of task
```

HPX equivalent:

```
#include <hpx/future.hpp>

```

(continues on next page)

(continued from previous page)

```
hpx::async([](){
    // task code
}).get(); // wait for the task to complete

// code after completion of task
```

The *get()* function can be used to ensure that the task created with `hpx::async` is completed before the code continues executing beyond that point.

Multiple tasks synchronization

OpenMP code:

```
#pragma omp task
{
    // task 1 code
}

#pragma omp task
{
    // task 2 code
}

#pragma omp taskwait
// code after completion of both tasks 1 and 2
```

HPX equivalent:

```
#include <hpx/future.hpp>

auto future1 = hpx::async([](){
    // task 1 code
});

auto future2 = hpx::async([](){
    // task 2 code
});

auto future = hpx::when_all(future1, future2).then([](auto&&){
    // code after completion of both tasks 1 and 2
});
```

If you would like to synchronize multiple tasks, you can use the `hpx::when_all` function to define which futures have to be ready and the *then()* function to declare what should be executed once these futures are ready.

Dependencies

OpenMP code:

```
int a = 10;
int b = 20;
int c = 0;

#pragma omp task depend(in: a, b) depend(out: c)
{
    // task code
    c = 100;
}
```

HPX equivalent:

```
#include <hpx/future.hpp>

int a = 10;
int b = 20;
int c = 0;

// Create a future representing 'a'
auto future_a = hpx::make_ready_future(a);

// Create a future representing 'b'
auto future_b = hpx::make_ready_future(b);

// Create a task that depends on 'a' and 'b' and executes 'task_code'
auto future_c = hpx::dataflow(
    []() {
        // task code
        return 100;
    },
    future_a, future_b);

c = future_c.get();
```

If one of the arguments of `hpx::dataflow` is a future, then it will wait for the future to be ready to launch the thread. Hence, to define the dependencies of tasks you have to create futures representing the variables that create dependencies and pass them as arguments to `hpx::dataflow`. *get()* is used to save the result of the future to the desired variable.

Nested tasks

OpenMP code:

```
#pragma omp task
{
    // Outer task code
    #pragma omp task
    {
        // Inner task code
```

(continues on next page)

(continued from previous page)

```
    }
}
```

HPX equivalent:

```
#include <hpx/future.hpp>

auto future_outer = hpx::async([](){
    // Outer task code

    hpx::async([](){
        // Inner task code
    });
});
```

or

```
#include <hpx/future.hpp>

auto future_outer = hpx::post([](){ // fire and forget
    // Outer task code

    hpx::post([](){ // fire and forget
        // Inner task code
    });
});
```

If you have nested tasks, you can simply use nested `hpx::async` or `hpx::post` calls. The implementation is similar if you want to take care of synchronization:

OpenMP code:

```
#pragma omp taskwait
{
    // Outer task code
    #pragma omp taskwait
    {
        // Inner task code
    }
}
```

HPX equivalent:

```
#include <hpx/future.hpp>

auto future_outer = hpx::async([]() {
    // Outer task code

    hpx::async([]() {
        // Inner task code
    }).get();    // Wait for the inner task to complete
});

future_outer.get();    // Wait for the outer task to complete
```

Task yield

OpenMP code:

```
#pragma omp task
{
    // code before yielding
    #pragma omp taskyield
    // code after yielding
}
```

HPX equivalent:

```
#include <hpx/future.hpp>
#include <hpx/thread.hpp>

auto future = hpx::async([](){
    // code before yielding
});

// yield execution to potentially allow other tasks to run
hpx::this_thread::yield();

// code after yielding
```

After creating a task using `hpx::async`, `hpx::this_thread::yield` can be used to reschedule the execution of threads, allowing other threads to run.

Task group

OpenMP code:

```
#pragma omp taskgroup
{
    #pragma omp task
    {
        // task 1 code
    }

    #pragma omp task
    {
        // task 2 code
    }
}
```

HPX equivalent:

```
#include <hpx/task_group.hpp>

// Declare a task group
hpx::experimental::task_group tg;

// Run the tasks
```

(continues on next page)

(continued from previous page)

```
tg.run([](){
    // task 1 code
});
tg.run(
    // task 2 code
});

// Wait for the task group
tg.wait();
```

To create task groups, you can use `hpx::experimental::task_group`. The function *run()* can be used to run each task within the task group, while *wait()* can be used to achieve synchronization. If you do not care about waiting for the task group to complete its execution, you can simply remove the *wait()* function.

OpenMP sections

OpenMP code:

```
#pragma omp sections
{
    #pragma omp section
    // section 1 code
    #pragma omp section
    // section 2 code
} // implicit synchronization
```

HPX equivalent:

```
#include <hpx/future.hpp>

auto future_section1 = hpx::async([](){
    // section 1 code
});
auto future_section2 = hpx::async([](){
    // section 2 code
);

// synchronization: wait for both sections to complete
hpx::wait_all(future_section1, future_section2);
```

Unlike tasks, there is an implicit synchronization barrier at the end of each *sections*` directive in OpenMP. This synchronization is achieved using `hpx::wait_all` function.

Note: If the *nowait* clause is used in the *sections* directive, then you can just remove the `hpx::wait_all` function while keeping the rest of the code as it is.

Intel Threading Building Blocks (TBB)

Intel Threading Building Blocks (TBB) provides a high-level interface for parallelism and concurrent programming using standard ISO C++ code. Below are some examples on how to convert Intel Threading Building Blocks (TBB) to *HPX* code:

parallel_for

Intel Threading Building Blocks (TBB) code:

```
auto values = std::vector<double>(10000);

tbb::parallel_for( tbb::blocked_range<int>(0,values.size()),
                   [&](tbb::blocked_range<int> r)
{
    for (int i=r.begin(); i<r.end(); ++i)
    {
        // loop body
    }
});
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>

auto values = std::vector<double>(10000);

hpx::experimental::for_loop(hpx::execution::par, 0, values.size(), [&](int i) {
    // loop body
});
```

In the above code, *tbb::parallel_for* is replaced with `hpx::experimental::for_loop` from the *HPX* library. The loop body within the lambda function will be executed in parallel for each iteration.

parallel_for_each

Intel Threading Building Blocks (TBB) code:

```
auto values = std::vector<double>(10000);

tbb::parallel_for_each(values.begin(), values.end(), [&](){
    // loop body
});
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>

auto values = std::vector<double>(10000);

hpx::for_each(hpx::execution::par, values.begin(), values.end(), [&](){
```

(continues on next page)

(continued from previous page)

```
    // loop body
});
```

By utilizing `hpx::for_each`` and specifying a parallel execution policy with *hpx::execution::par*, it is possible to transform *tbb::parallel_for_each`* into its equivalent counterpart in *HPX*.

parallel_invoke

Intel Threading Building Blocks (TBB) code:

```
tbb::parallel_invoke(task1, task2, task3);
```

HPX equivalent:

```
#include <hpx/future.hpp>

hpx::wait_all(hpx::async(task1), hpx::async(task2), hpx::async(task3));
```

To convert *tbb::parallel_invoke* to *HPX*, we use `hpx::async` to asynchronously execute each task, which returns a future representing the result of each task. We then pass these futures to `hpx::when_all`, which waits for all the futures to complete before returning.

parallel_pipeline

Intel Threading Building Blocks (TBB) code:

```
tbb::parallel_pipeline(4,
    tbb::make_filter<void, int>(tbb::filter::serial_in_order,
        [](tbb::flow_control& fc) -> int {
            // Generate numbers from 1 to 10
            static int i = 1;
            if (i <= 10) {
                return i++;
            }
            else {
                fc.stop();
                return 0;
            }
        }) &
    tbb::make_filter<int, int>(tbb::filter::parallel,
        [](int num) -> int {
            // Multiply each number by 2
            return num * 2;
        }) &
    tbb::make_filter<int, void>(tbb::filter::serial_in_order,
        [](int num) {
            // Print the results
            std::cout << num << " ";
        })
);
```

HPX equivalent:

```
#include <iostream>
#include <vector>
#include <ranges>
#include <hpx/algorithm.hpp>

// generate the values
auto range = std::views::iota(1) | std::views::take(10);

// materialize the output vector
std::vector<int> results(10);

// in parallel execution of pipeline and transformation
hpx::ranges::transform(
    hpx::execution::par, range, result.begin(), [](int i) { return 2 * i; });

// print the modified vector
for (int i : result)
{
    std::cout << i << " ";
}
std::cout << std::endl;
```

The line *auto range = std::views::iota(1) | std::views::take(10);* generates a range of values using the *std::views::iota* function. It starts from the value 1 and generates an infinite sequence of incrementing values. The *std::views::take(10)* function is then applied to limit the sequence to the first 10 values. The result is stored in the *range* variable.

Hint: A view is a lightweight object that represents a particular view of a sequence or range. It acts as a read-only interface to the original data, providing a way to query and traverse the elements without making any copies or modifications.

Views can be composed and chained together to form complex pipelines of operations. These operations are evaluated lazily, meaning that the actual computation is performed only when the result is needed or consumed.

Since views perform lazy evaluation, we use *std::vector<int> results(10);* to meterialize the vector that will store the transformed values. The *hpx::ranges::transform* function is then used to perform a parallel transformation on the range. The transformed values will be written to the *results* vector.

Hint: Ranges enable loop fusion by combining multiple operations into a single parallel loop, eliminating waiting time and reducing overhead. Using ranges, you can express these operations as a pipeline of transformations on a sequence of elements. This pipeline is evaluated in a single pass, performing all the desired operations in parallel without the need to wait between them.

In addition, *HPX* enhances the benefits of range fusion by offering parallel execution policies, which can be used to optimize the execution of the fused loop across multiple threads.

---

parallel_reduce

Reduction

Intel Threading Building Blocks (TBB) code:

```
auto values = std::vector<double>{1,2,3,4,5,6,7,8,9};

auto total = tbb::parallel_reduce(
                tbb::blocked_range<int>(0,values.size()),
                0.0,
                [&](tbb::blocked_range<int> r, double running_total)
                {
                    for (int i=r.begin(); i<r.end(); ++i)
                    {
                        running_total += values[i];
                    }


                    return running_total;
                },
                std::plus<double>());
```

HPX equivalent:

```
#include <hpx/numeric.hpp>

auto values = std::vector<double>{1,2,3,4,5,6,7,8,9};

auto total = hpx::reduce(
    hpx::execution::par, values.begin(), values.end(), 0, std::plus{});
```

By utilizing `hpx::reduce` and specifying a parallel execution policy with *hpx::execution::par*, it is possible to transform *tbb::parallel_reduce*` into its equivalent counterpart in *HPX*. As demonstrated in the previous example, the management of intermediate results is seamlessly handled internally by *HPX*, eliminating the need for explicit consideration.

Transformation & Reduction

Intel Threading Building Blocks (TBB) code:

```
auto values = std::vector<double>{1,2,3,4,5,6,7,8,9};

auto transform_function(double current_value){
    // transformation code
}

auto total = tbb::parallel_reduce(
                tbb::blocked_range<int>(0,values.size()),
                0.0,
                [&](tbb::blocked_range<int> r, double transformed_val)
                {
                    for (int i=r.begin(); i<r.end(); ++i)
```

(continues on next page)

(continued from previous page)

```
                    {
                        transformed_val += transform_function(values[i]);
                    }
                    return transformed_val;
                },
                std::plus<double>());
```

HPX equivalent:

```
#include <hpx/numeric.hpp>

auto values = std::vector<double>{1,2,3,4,5,6,7,8,9};

auto transform_function(double current_value)
{
    // transformation code
}

auto total = hpx::transform_reduce(hpx::execution::par, values.begin(),
    values.end(), 0, std::plus{},
    [&](double current_value) { return transform_function(current_value); });
```

In situations where certain values require transformation before the reduction process, *HPX* provides a straightforward solution through `hpx::transform_reduce`. The *transform_function()* allows for the application of the desired transformation to each value.

parallel_scan

Intel Threading Building Blocks (TBB) code:

```
tbb::parallel_scan(tbb::blocked_range<size_t>(0, input.size()),
    0,
    [&input, &output](const tbb::blocked_range<size_t>& range, int& partial_sum, bool is_
↪final_scan) {
        for (size_t i = range.begin(); i != range.end(); ++i) {
            partial_sum += input[i];
            if (is_final_scan) {
                output[i] = partial_sum;
            }
        }
        return partial_sum;
    },
    [](int left_sum, int right_sum) {
        return left_sum + right_sum;
    }
);
```

HPX equivalent:

```
#include <hpx/numeric.hpp>

hpx::inclusive_scan(hpx::execution::par, input.begin(), input.end(),
```

(continues on next page)

(continued from previous page)

```
    output.begin(),
    [](const int& left, const int& right) { return left + right; });
```

`hpx::inclusive_scan` with *hpx::execution::par* as execution policy can be used to perform a prefix scan in parallel. The management of intermediate results is seamlessly handled internally by *HPX*, eliminating the need for explicit consideration. *input.begin()* and *input.end()* refer to the beginning and end of the sequence of elements the algorithm will be applied to respectively. *output.begin()* refers to the beginning of the destination, while the last argument specifies the function which will be invoked for each of the values of the input sequence.

Apart from `hpx::inclusive_scan`, *HPX* provides its users with `hpx::exclusive_scan`. The key difference between inclusive scan and exclusive scan lies in the treatment of the current element during the scan operation. In an inclusive scan, each element in the output sequence includes the contribution of the corresponding element in the input sequence, while in an exclusive scan, the current element in the input sequence does not contribute to the corresponding element in the output sequence.

parallel_sort

Intel Threading Building Blocks (TBB) code:

```
std::vector<int> numbers = {9, 2, 7, 1, 5, 3};

tbb::parallel_sort(numbers.begin(), numbers.end());
```

HPX equivalent:

```
#include <hpx/algorithm.hpp>

std::vector<int> numbers = {9, 2, 7, 1, 5, 3};

hpx::sort(hpx::execution::par, numbers.begin(), numbers.end());
```

`hpx::sort` provides an equivalent functionality to *tbb::parallel_sort*. When given a parallel execution policy with *hpx::execution::par*, the algorithm employs parallel execution, allowing for efficient sorting across available threads.

task_group

Intel Threading Building Blocks (TBB) code:

```
// Declare a task group
tbb::task_group tg;

// Run the tasks
tg.run(task1);
tg.run(task2);

// Wait for the task group
tg.wait();
```

HPX equivalent:

```
#include <hpx/task_group.hpp>

// Declare a task group
hpx::experimental::task_group tg;

// Run the tasks
tg.run(task1);
tg.run(task2);

// Wait for the task group
tg.wait();
```

HPX drew inspiration from Intel Threading Building Blocks (TBB) to introduce the `hpx::experimental::task_group` feature. Therefore, utilizing `hpx::experimental::task_group` provides an equivalent functionality to *tbb::task_group*.

MPI

MPI is a standardized communication protocol and library that allows multiple processes or nodes in a parallel computing system to exchange data and coordinate their execution.

List of MPI-*HPX* functions

MPI function	*HPX* equivalent
MPI_Allgather	`hpx::collectives::all_gather`
MPI_Allreduce	`hpx::collectives::all_reduce`
MPI_Alltoall	`hpx::collectives::all_to_all`
MPI_Barrier	`hpx::distributed::barrier`
MPI_Bcast	`hpx::collectives::broadcast_to()` and `hpx::collectives::broadcast_from()` used with `get()`
MPI_Comm_size	`hpx::get_num_localities`
MPI_Comm_rank	`hpx::get_locality_id()`
MPI_Exscan	`hpx::collectives::exclusive_scan()` used with `get()`
MPI_Gather	`hpx::collectives::gather_here()` and `hpx::collectives::gather_there()` used with `get()`
MPI_Irecv	`hpx::collectives::get()`
MPI_Isend	`hpx::collectives::set()`
MPI_Reduce	`hpx::collectives::reduce_here` and `hpx::collectives::reduce_there` used with `get()`
MPI_Scan	`hpx::collectives::inclusive_scan()` used with `get()`
MPI_Scatter	`hpx::collectives::scatter_to()` and `hpx::collectives::scatter_from()`
MPI_Wait	`hpx::collectives::get()` used with a future i.e. `setf.get()`

MPI_Send & MPI_Recv

Let's assume we have the following simple message passing code where each process sends a message to the next process in a circular manner. The exchanged message is modified and printed to the console.

MPI code:

```
#include <cstddef>
#include <cstdint>
#include <iostream>
#include <mpi.h>
#include <vector>

constexpr int times = 2;

int main(int argc, char *argv[]) {
MPI_Init(&argc, &argv);

int num_localities;
MPI_Comm_size(MPI_COMM_WORLD, &num_localities);

int this_locality;
MPI_Comm_rank(MPI_COMM_WORLD, &this_locality);

int next_locality = (this_locality + 1) % num_localities;
std::vector<int> msg_vec = {0, 1};

int cnt = 0;
int msg = msg_vec[this_locality];

int recv_msg;
MPI_Request request_send, request_recv;
MPI_Status status;

while (cnt < times) {
    cnt += 1;

    MPI_Isend(&msg, 1, MPI_INT, next_locality, cnt, MPI_COMM_WORLD,
            &request_send);
    MPI_Irecv(&recv_msg, 1, MPI_INT, next_locality, cnt, MPI_COMM_WORLD,
            &request_recv);

    MPI_Wait(&request_send, &status);
    MPI_Wait(&request_recv, &status);

    std::cout << "Time: " << cnt << ", Locality " << this_locality
            << " received msg: " << recv_msg << "\n";

    recv_msg += 10;
    msg = recv_msg;
}

MPI_Finalize();
```

(continues on next page)

(continued from previous page)

```
return 0;
}
```

HPX equivalent:

```
#include <hpx/config.hpp>

#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/algorithm.hpp>
#include <hpx/hpx_init.hpp>
#include <hpx/modules/collectives.hpp>

#include <cstddef>
#include <cstdint>
#include <iostream>
#include <utility>
#include <vector>

using namespace hpx::collectives;

constexpr char const* channel_communicator_name =
    "/example/channel_communicator/";

// the number of times
constexpr int times = 2;

int hpx_main()
{
    std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
    std::uint32_t this_locality = hpx::get_locality_id();

    // allocate channel communicator
    auto comm = create_channel_communicator(hpx::launch::sync,
        channel_communicator_name, num_sites_arg(num_localities),
        this_site_arg(this_locality));

    std::uint32_t next_locality = (this_locality + 1) % num_localities;
    std::vector<int> msg_vec = {0, 1};

    int cnt = 0;
    int msg = msg_vec[this_locality];

    // send values to another locality
    auto setf = set(comm, that_site_arg(next_locality), msg, tag_arg(cnt));
    auto got_msg = get<int>(comm, that_site_arg(next_locality), tag_arg(cnt));

    setf.get();

    while (cnt < times)
    {
        cnt += 1;
```

(continues on next page)

(continued from previous page)

```
        auto done_msg = got_msg.then([&](auto&& f) {
            int rec_msg = f.get();
            std::cout << "Time: " << cnt << ", Locality " << this_locality
                      << " received msg: " << rec_msg << "\n";

            // change msg by adding 10
            rec_msg += 10;

            // start next round
            setf =
                set(comm, that_site_arg(next_locality), rec_msg, tag_arg(cnt));
            got_msg =
                get<int>(comm, that_site_arg(next_locality), tag_arg(cnt));
            setf.get();
        });

        done_msg.get();
    }

    return hpx::finalize();
}
#endif

int main(int argc, char* argv[])
{
#if !defined(HPX_COMPUTE_DEVICE_CODE)
    hpx::init_params params;
    params.cfg = {"--hpx:run-hpx-main"};
    return hpx::init(argc, argv, params);
#else
    (void) argc;
    (void) argv;
    return 0;
#endif
}
```

To perform message passing between different processes in *HPX* we can use a channel communicator. To understand this example, let's focus on the *hpx_main()* function:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- *create_channel_communicator* function is used to create a channel to serve the communication. This function takes several arguments, including the launch policy (*hpx::launch::sync*), the name of the communicator (*channel_communicator_name*), the number of localities, and the ID of the current locality.
- The communication follows a ring pattern, where each process (or locality) sends a message to its neighbor in a circular manner. This means that the messages circulate around the localities, ensuring that the communication wraps around when reaching the end of the locality sequence. To achieve this, the *next_locality* variable is calculated as the ID of the next locality in the ring.
- The initial values for the communication are set (*msg_vec*, *cnt*, *msg*).
- The *set()* function is called to send the message to the next locality in the ring. The message is sent asynchronously and is associated with a tag (*cnt*).

- The *get()* function is called to receive a message from the next locality. It is also associated with the same tag as the *set()* operation.
- The *setf.get()* call blocks until the message sending operation is complete.
- A continuation is set up using the function *then()* to handle the received message. Inside the continuation:
  - The received message value (*rec_msg*) is retrieved using *f.get()*.
  - The received message is printed to the console and then modified by adding 10.
  - The *set()* and *get()* operations are repeated to send and receive the modified message to the next locality.
  - The *setf.get()* call blocks until the new message sending operation is complete.
- The *done_msg.get()* call blocks until the continuation is complete for the current loop iteration.

Having said that, we conclude to the following table:

MPI_Gather

The following code gathers data from all processes to the root process and verifies the gathered data in the root process.

MPI code:

```
#include <iostream>
#include <mpi.h>
#include <numeric>
#include <vector>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int num_localities, this_locality;
    MPI_Comm_size(MPI_COMM_WORLD, &num_localities);
    MPI_Comm_rank(MPI_COMM_WORLD, &this_locality);

    std::vector<int> local_data; // Data to be gathered

    if (this_locality == 0) {
        local_data.resize(num_localities); // Resize the vector on the root process
    }

    // Each process calculates its local data value
    int my_data = 42 + this_locality;

    for (std::uint32_t i = 0; i != 10; ++i) {

        // Gather data from all processes to the root process (process 0)
        MPI_Gather(&my_data, 1, MPI_INT, local_data.data(), 1, MPI_INT, 0,
                MPI_COMM_WORLD);

        // Only the root process (process 0) will print the gathered data
        if (this_locality == 0) {
        std::cout << "Gathered data on the root: ";
        for (int i = 0; i < num_localities; ++i) {
            std::cout << local_data[i] << " ";
```

(continues on next page)

(continued from previous page)

```
        }
        std::cout << std::endl;
        }
    }
    std::cout << std::endl;

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
std::uint32_t this_locality = hpx::get_locality_id();

// test functionality based on immediate local result value
auto gather_direct_client = create_communicator(gather_direct_basename,
    num_sites_arg(num_localities), this_site_arg(this_locality));

for (std::uint32_t i = 0; i != 10; ++i)
{
    if (this_locality == 0)
    {
        hpx::future<std::vector<std::uint32_t>> overall_result =
            gather_here(gather_direct_client, std::uint32_t(42));

        std::vector<std::uint32_t> sol = overall_result.get();
        std::cout << "Gathered data on the root:";

        for (std::size_t j = 0; j != sol.size(); ++j)
        {
            HPX_TEST(j + 42 == sol[j]);
            std::cout << " " << sol[j];
        }
        std::cout << std::endl;
    }
    else
    {
        hpx::future<void> overall_result =
            gather_there(gather_direct_client, this_locality + 42);
        overall_result.get();
    }

}
```

This code will print 10 times the following message:

```
Gathered data on the root: 42 43
```

HPX uses two functions to implement the functionality of *MPI_Gather*: *gather_here* and *gather_there*. *gather_here* is gathering data from all localities to the locality with ID 0 (root locality). *gather_there* allows non-root localities to participate in the gather operation by sending data to the root locality. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.

- The function *create_communicator()* is used to create a communicator called *gather_direct_client*.
- If the current locality is the root (its ID is equal to 0):
  - The *gather_here* function is used to perform the gather operation. It collects data from all other localities into the *overall_result* future object. The function arguments provide the necessary information, such as the base name for the gather operation (*gather_direct_basename*), the value to be gathered (*value*), the number of localities (*num_localities*), the current locality ID (*this_locality*), and the generation number (related to the gather operation).
  - The *get()* member function of the *overall_result* future is used to retrieve the gathered data.
  - The next *for* loop is used to verify the correctness of the gathered data (*sol*). *HPX_TEST* is a macro provided by the *HPX* testing utilities to perform similar testing with the Standard C++ macro *assert*.
- If the current locality is not the root:
  - The *gather_there* function is used to participate in the gather operation initiated by the root locality. It sends the data (in this case, the value *this_locality + 42*) to the root locality, indicating that it should be included in the gathering.
  - The *get()* member function of the *overall_result* future is used to wait for the gather operation to complete for this locality.

MPI_Scatter

The following code gathers data from all processes to the root process and verifies the gathered data in the root process.

MPI code:

```
#include <iostream>
#include <mpi.h>
#include <vector>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int num_localities, this_locality;
    MPI_Comm_size(MPI_COMM_WORLD, &num_localities);
    MPI_Comm_rank(MPI_COMM_WORLD, &this_locality);

    int num_localities = num_localities;
    std::vector<int> data(num_localities);

    if (this_locality == 0) {
        // Fill the data vector on the root locality (locality 0)
        for (int i = 0; i < num_localities; ++i) {
        data[i] = 42 + i;
        }
    }

    int local_data; // Variable to store the received data

    // Scatter data from the root locality to all other localities
    MPI_Scatter(&data[0], 1, MPI_INT, &local_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

(continues on next page)

(continued from previous page)

```
    // Now, each locality has its own local_data

    // Print the local_data on each locality
    std::cout << "Locality " << this_locality << " received " << local_data
              << std::endl;

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
HPX_TEST_LTE(std::uint32_t(2), num_localities);

std::uint32_t this_locality = hpx::get_locality_id();

auto scatter_direct_client =
    hpx::collectives::create_communicator(scatter_direct_basename,
        num_sites_arg(num_localities), this_site_arg(this_locality));

// test functionality based on immediate local result value
for (std::uint32_t i = 0; i != 10; ++i)
{
    if (this_locality == 0)
    {
        std::vector<std::uint32_t> data(num_localities);
        std::iota(data.begin(), data.end(), 42 + i);

        hpx::future<std::uint32_t> result =
            scatter_to(scatter_direct_client, std::move(data));

        HPX_TEST_EQ(i + 42 + this_locality, result.get());
    }
    else
    {
        hpx::future<std::uint32_t> result =
            scatter_from<std::uint32_t>(scatter_direct_client);

        HPX_TEST_EQ(i + 42 + this_locality, result.get());

        std::cout << "Locality " << this_locality << " received "
                  << i + 42 + this_locality << std::endl;
    }
}
```

For num_localities = 2 and since we run for 10 iterations this code will print the following message:

```
Locality 1 received 43
Locality 1 received 44
Locality 1 received 45
```

(continues on next page)

(continued from previous page)

```
Locality 1 received 46
Locality 1 received 47
Locality 1 received 48
Locality 1 received 49
Locality 1 received 50
Locality 1 received 51
Locality 1 received 52
```

HPX uses two functions to implement the functionality of *MPI_Scatter*: *hpx::scatter_to* and *hpx::scatter_from*. *hpx::scatter_to* is distributing the data from the locality with ID 0 (root locality) to all other localities. *hpx::scatter_from* allows non-root localities to receive the data from the root locality. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- The function *hpx::collectives::create_communicator()* is used to create a communicator called *scatter_direct_client*.
- If the current locality is the root (its ID is equal to 0):
  - The data vector is filled with values ranging from *42 + i* to *42 + i + num_localities - 1*.
  - The *hpx::scatter_to* function is used to perform the scatter operation using the communicator *scatter_direct_client*. This scatters the data vector to other localities and returns a future representing the result.
  - *HPX_TEST_EQ* is a macro provided by the *HPX* testing utilities to test the distributed values.
- If the current locality is not the root:
  - The *hpx::scatter_from* function is used to collect the data by the root locality.
  - *HPX_TEST_EQ* is a macro provided by the *HPX* testing utilities to test the collected values.

MPI_Allgather

The following code gathers data from all processes and sends the data to all processes.

MPI code:

```
#include <cstdint>
#include <iostream>
#include <mpi.h>
#include <vector>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the number of MPI processes
    int num_localities = size;

    // Get the MPI process rank
    int here = rank;
```

(continues on next page)

(continued from previous page)

```
    std::uint32_t value = here;

    std::vector<std::uint32_t> r(num_localities);

    // Perform an all-gather operation to gather values from all processes.
    MPI_Allgather(&value, 1, MPI_UINT32_T, r.data(), 1, MPI_UINT32_T,
                  MPI_COMM_WORLD);

    // Print the result.
    std::cout << "Locality " << here << " has values:";
    for (size_t j = 0; j < r.size(); ++j) {
        std::cout << " " << r[j];
    }
    std::cout << std::endl;

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
std::uint32_t here = hpx::get_locality_id();

// test functionality based on immediate local result value
auto all_gather_direct_client =
    create_communicator(all_gather_direct_basename,
        num_sites_arg(num_localities), this_site_arg(here));

std::uint32_t value = here;

hpx::future<std::vector<std::uint32_t>> overall_result =
    all_gather(all_gather_direct_client, value);

std::vector<std::uint32_t> r = overall_result.get();

std::cout << "Locality " << here << " has values:";
for (std::size_t j = 0; j != r.size(); ++j)
{
    std::cout << " " << j;
}
std::cout << std::endl;
```

For num_localities = 2 this code will print the following message:

```
Locality 0 has values: 0 1
Locality 1 has values: 0 1
```

HPX uses the function *all_gather* to implement the functionality of *MPI_Allgather*. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.

- The function *hpx::collectives::create_communicator()* is used to create a communicator called *all_gather_direct_client*.
- The values that the localities exchange with each other are equal to each locality's ID.
- The gather operation is performed using *all_gather*. The result is stored in an *hpx::future* object called *overall_result*, which represents a future result that can be retrieved later when needed.
- The *get()* function waits until the result is available and then stores it in the vector called *r*.

MPI_Allreduce

The following code combines values from all processes and distributes the result back to all processes.

MPI code:

```
#include <cstdint>
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the number of MPI processes
    int num_localities = size;

    // Get the MPI process rank
    int here = rank;

    // Create a communicator for the all reduce operation.
    MPI_Comm all_reduce_direct_client;
    MPI_Comm_split(MPI_COMM_WORLD, 0, rank, &all_reduce_direct_client);

    // Perform the all reduce operation to calculate the sum of 'here' values.
    std::uint32_t value = here;
    std::uint32_t res = 0;
    MPI_Allreduce(&value, &res, 1, MPI_UINT32_T, MPI_SUM,
                    all_reduce_direct_client);

    std::cout << "Locality " << rank << " has value: " << res << std::endl;

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t const num_localities =
    hpx::get_num_localities(hpx::launch::sync);
std::uint32_t const here = hpx::get_locality_id();
```

(continues on next page)

(continued from previous page)

```
auto const all_reduce_direct_client =
    create_communicator(all_reduce_direct_basename,
        num_sites_arg(num_localities), this_site_arg(here));

std::uint32_t value = here;

hpx::future<std::uint32_t> overall_result =
    all_reduce(all_reduce_direct_client, value, std::plus<std::uint32_t>{});

std::uint32_t res = overall_result.get();
std::cout << "Locality " << here << " has value: " << res << std::endl;
```

For num_localities = 2 this code will print the following message:

```
Locality 0 has value: 1
Locality 1 has value: 1
```

HPX uses the function *all_reduce* to implement the functionality of *MPI_Allreduce*. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- The function *hpx::collectives::create_communicator()* is used to create a communicator called *all_reduce_direct_client*.
- The value of each locality is equal to its ID.
- The reduce operation is performed using *all_reduce*. The result is stored in an *hpx::future* object called *overall_result*, which represents a future result that can be retrieved later when needed.
- The *get()* function waits until the result is available and then stores it in the variable *res*.

MPI_Alltoall

The following code gathers data from and scatters data to all processes.

MPI code:

```
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <mpi.h>
#include <vector>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the number of MPI processes
    int num_localities = size;

```

(continues on next page)

(continued from previous page)

```
    // Get the MPI process rank
    int this_locality = rank;

    // Create a communicator for all-to-all operation.
    MPI_Comm all_to_all_direct_client;
    MPI_Comm_split(MPI_COMM_WORLD, 0, rank, &all_to_all_direct_client);

    std::vector<std::uint32_t> values(num_localities);
    std::fill(values.begin(), values.end(), this_locality);

    // Create vectors to store received values.
    std::vector<std::uint32_t> r(num_localities);

    // Perform an all-to-all operation to exchange values with other localities.
    MPI_Alltoall(values.data(), 1, MPI_UINT32_T, r.data(), 1, MPI_UINT32_T,
                 all_to_all_direct_client);

    // Print the results.
    std::cout << "Locality " << this_locality << " has values:";
    for (std::size_t j = 0; j != r.size(); ++j) {
        std::cout << " " << r[j];
    }
    std::cout << std::endl;

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
std::uint32_t this_locality = hpx::get_locality_id();

auto all_to_all_direct_client =
    create_communicator(all_to_all_direct_basename,
        num_sites_arg(num_localities), this_site_arg(this_locality));

std::vector<std::uint32_t> values(num_localities);
std::fill(values.begin(), values.end(), this_locality);

hpx::future<std::vector<std::uint32_t>> overall_result =
    all_to_all(all_to_all_direct_client, std::move(values));

std::vector<std::uint32_t> r = overall_result.get();
std::cout << "Locality " << this_locality << " has values:";

for (std::size_t j = 0; j != r.size(); ++j)
{
    std::cout << " " << r[j];
}
std::cout << std::endl;
```

For num_localities = 2 this code will print the following message:

```
Locality 0 has values: 0 1
Locality 1 has values: 0 1
```

HPX uses the function *all_to_all* to implement the functionality of *MPI_Alltoall*. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- The function *hpx::collectives::create_communicator()* is used to create a communicator called *all_to_all_direct_client*.
- The value each locality sends is equal to its ID.
- The all-to-all operation is performed using *all_to_all*. The result is stored in an *hpx::future* object called *overall_result*, which represents a future result that can be retrieved later when needed.
- The *get()* function waits until the result is available and then stores it in the variable *r*.

MPI_Barrier

The following code shows how barrier is used to synchronize multiple processes.

MPI code:

```
#include <cstdlib>
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    std::size_t iterations = 5;

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    for (std::size_t i = 0; i != iterations; ++i) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (rank == 0) {
        std::cout << "Iteration " << i << " completed." << std::endl;
        }
    }

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::size_t iterations = 5;
std::uint32_t this_locality = hpx::get_locality_id();

char const* const barrier_test_name = "/test/barrier/multiple";

```

(continues on next page)

(continued from previous page)

```
hpx::distributed::barrier b(barrier_test_name);
for (std::size_t i = 0; i != iterations; ++i)
{
    b.wait();
    if (this_locality == 0)
    {
        std::cout << "Iteration " << i << " completed." << std::endl;
    }
}
```

This code will print the following message:

```
Iteration 0 completed.
Iteration 1 completed.
Iteration 2 completed.
Iteration 3 completed.
Iteration 4 completed.
```

HPX uses the function *barrier* to implement the functionality of *MPI_Barrier*. In more detail:

- After defining the number of iterations, we use *hpx::get_locality_id()* to get the ID of the current locality.
- *char const* const barrier_test_name = "/test/barrier/multiple"*: This line defines a constant character array as the name of the barrier. This name is used to identify the barrier across different localities. All participating threads that use this name will synchronize at this barrier.
- Using *hpx::distributed::barrier b(barrier_test_name)*, we create an instance of the distributed barrier with the previously defined name. This barrier will be used to synchronize the execution of threads across different localities.
- Running for all the desired iterations, we use *b.wait()* to synchronize the threads. Each thread waits until all other threads also reach this point before any of them can proceed further.

MPI_Bcast

The following code broadcasts data from one process to all other processes.

MPI code:

```
#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int num_localities;
    MPI_Comm_size(MPI_COMM_WORLD, &num_localities);

    int here;
    MPI_Comm_rank(MPI_COMM_WORLD, &here);

    int value;

    for (int i = 0; i < 5; ++i) {
```

(continues on next page)

(continued from previous page)

```
        if (here == 0) {
            value = i + 42;
        }

        // Broadcast the value from process 0 to all other processes
        MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if (here != 0) {
            std::cout << "Locality " << here << " received " << value << std::endl;
        }

    }

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);

std::uint32_t here = hpx::get_locality_id();

auto broadcast_direct_client =
    create_communicator(broadcast_direct_basename,
        num_sites_arg(num_localities), this_site_arg(here));

// test functionality based on immediate local result value
for (std::uint32_t i = 0; i != 5; ++i)
{
    if (here == 0)
    {
        hpx::future<std::uint32_t> result =
            broadcast_to(broadcast_direct_client, i + 42);

        result.get();
    }
    else
    {
        hpx::future<std::uint32_t> result =
            hpx::collectives::broadcast_from<std::uint32_t>(
                broadcast_direct_client);

        uint32_t r = result.get();

        std::cout << "Locality " << here << " received " << r << std::endl;
    }
}
```

For num_localities = 2 this code will print the following message:

```
Locality 1 received 42
```

(continues on next page)

(continued from previous page)

```
Locality 1 received 43
Locality 1 received 44
Locality 1 received 45
Locality 1 received 46
```

HPX uses two functions to implement the functionality of *MPI_Bcast*: *broadcast_to* and *broadcast_from*. *broadcast_to* is broadcasting the data from the root locality to all other localities. *broadcast_from* allows non-root localities to collect the data sent by the root locality. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- The function *create_communicator()* is used to create a communicator called *broadcast_direct_client*.
- If the current locality is the root (its ID is equal to 0):
  - The *broadcast_to* function is used to perform the broadcast operation using the communicator *broadcast_direct_client*. This sends the data to other localities and returns a future representing the result.
  - The *get()* member function of the *result* future is used to wait for and retrieve the result.
- If the current locality is not the root:
  - The *broadcast_from* function is used to collect the data by the root locality.
  - The *get()* member function of the *result* future is used to wait for the result.

MPI_Exscan

The following code computes the exclusive scan (partial reductions) of data on a collection of processes.

MPI code:

```
#include <iostream>
#include <mpi.h>
#include <numeric>
#include <vector>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int num_localities;
    MPI_Comm_size(MPI_COMM_WORLD, &num_localities);

    int here;
    MPI_Comm_rank(MPI_COMM_WORLD, &here);

    // Calculate the value for this locality (here)
    int value = here;

    // Perform an exclusive scan
    std::vector<int> result(num_localities);
    MPI_Exscan(&value, &result[0], 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    if (here != 0) {
        int r = result[here - 1]; // Result is in the previous rank's slot
```

(continues on next page)

(continued from previous page)

```
        std::cout << "Locality " << here << " has value " << r << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
std::uint32_t here = hpx::get_locality_id();

auto exclusive_scan_client = create_communicator(exclusive_scan_basename,
    num_sites_arg(num_localities), this_site_arg(here));

// test functionality based on immediate local result value
std::uint32_t value = here;

hpx::future<std::uint32_t> overall_result = exclusive_scan(
    exclusive_scan_client, value, std::plus<std::uint32_t>{});

uint32_t r = overall_result.get();

if (here != 0)
{
    std::cout << "Locality " << here << " has value " << r << std::endl;
}
```

For num_localities = 2 this code will print the following message:

```
Locality 1 has value 0
```

HPX uses the function *exclusive_scan* to implement *MPI_Exscan*. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- The function *create_communicator()* is used to create a communicator called *exclusive_scan_client*.
- The *exclusive_scan* function is used to perform the exclusive scan operation using the communicator *exclusive_scan_client*. *std::plus<std::uint32_t>{}* specifies the binary associative operator to use for the scan. In this case, it's addition for summing values.
- The *get()* member function of the *overall_result* future is used to wait for the result.

MPI_Scan

The following code Computes the inclusive scan (partial reductions) of data on a collection of processes.

MPI code:

```
#include <iostream>
#include <mpi.h>
#include <numeric>
#include <vector>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int num_localities;
    MPI_Comm_size(MPI_COMM_WORLD, &num_localities);

    int here;
    MPI_Comm_rank(MPI_COMM_WORLD, &here);

    // Calculate the value for this locality (here)
    int value = here;

    std::vector<int> result(num_localities);

    MPI_Scan(&value, &result[0], 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    std::cout << "Locality " << here << " has value " << result[0] << std::endl;

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
std::uint32_t here = hpx::get_locality_id();

auto inclusive_scan_client = create_communicator(inclusive_scan_basename,
    num_sites_arg(num_localities), this_site_arg(here));

std::uint32_t value = here;

hpx::future<std::uint32_t> overall_result = inclusive_scan(
    inclusive_scan_client, value, std::plus<std::uint32_t>{});

uint32_t r = overall_result.get();

std::cout << "Locality " << here << " has value " << r << std::endl;
```

For num_localities = 2 this code will print the following message:

```
Locality 0 has value 0
Locality 1 has value 1
```

HPX uses the function *inclusive_scan* to implement *MPI_Scan*. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- The function *create_communicator()* is used to create a communicator called *inclusive_scan_client*.
- The *inclusive_scan* function is used to perform the exclusive scan operation using the communicator *inclusive_scan_client*. *std::plus<std::uint32_t>{}* specifies the binary associative operator to use for the scan. In this case, it's addition for summing values.
- The *get()* member function of the *overall_result* future is used to wait for the result.

MPI_Reduce

The following code performs a global reduce operation across all processes.

MPI code:

```
#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int num_processes;
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);

    int this_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &this_rank);

    int value = this_rank;

    int result = 0;

    // Perform the reduction operation
    MPI_Reduce(&value, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print the result for the root process (process 0)
    if (this_rank == 0) {
        std::cout << "Locality " << this_rank << " has value " << result
                  << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

HPX equivalent:

```
std::uint32_t num_localities = hpx::get_num_localities(hpx::launch::sync);
std::uint32_t this_locality = hpx::get_locality_id();

auto reduce_direct_client = create_communicator(reduce_direct_basename,
    num_sites_arg(num_localities), this_site_arg(this_locality));
```

(continues on next page)

(continued from previous page)

```
std::uint32_t value = hpx::get_locality_id();

if (this_locality == 0)
{
    hpx::future<std::uint32_t> overall_result = reduce_here(
        reduce_direct_client, value, std::plus<std::uint32_t>{});

    uint32_t r = overall_result.get();

    std::cout << "Locality " << this_locality << " has value " << r
              << std::endl;
}
else
{
    hpx::future<void> overall_result =
        reduce_there(reduce_direct_client, std::move(value));
    overall_result.get();
}
```

This code will print the following message:

```
Locality 0 has value 1
```

HPX uses two functions to implement the functionality of *MPI_Reduce*: *reduce_here* and *reduce_there*. *reduce_here* is gathering data from all localities to the locality with ID 0 (root locality) and then performs the defined reduction operation. *reduce_there* allows non-root localities to participate in the reduction operation by sending data to the root locality. In more detail:

- *hpx::get_num_localities(hpx::launch::sync)* retrieves the number of localities, while *hpx::get_locality_id()* returns the ID of the current locality.
- The function *create_communicator()* is used to create a communicator called *reduce_direct_client*.
- If the current locality is the root (its ID is equal to 0):
  - The *reduce_here* function initiates a reduction operation with addition (*std::plus*) as the reduction operator. The result is stored in *overall_result*.
  - The *get()* member function of the *overall_result* future is used to wait for the result.
- If the current locality is not the root:
  - The *reduce_there* initiates a remote reduction operation.
  - The *get()* member function of the *overall_result* future is used to wait for the remote reduction operation to complete. This is done to ensure synchronization among localities.

2.3.6 Building tests and examples

Tests

To build the tests:

```
$ cmake --build . --target tests
```

To control which tests to run use `ctest`:

- To run single tests, for example a test for `for_loop`:

```
$ ctest --output-on-failure -R tests.unit.modules.algorithms.algorithms.for_loop
```

- To run a whole group of tests:

```
$ ctest --output-on-failure -R tests.unit
```

Examples

- To build (and install) all examples invoke:

```
$ cmake -DHPX_WITH_EXAMPLES=On .
$ make examples
$ make install
```

- To build the `hello_world_1` example run:

```
$ make hello_world_1
```

HPX executables end up in the `bin` directory in your build directory. You can now run `hello_world_1` and should see the following output:

```
$ ./bin/hello_world_1
Hello World!
```

You've just run an example which prints `Hello World!` from the *HPX* runtime. The source for the example is in `examples/quickstart/hello_world_1.cpp`. The `hello_world_distributed` example (also available in the `examples/quickstart` directory) is a distributed hello world program, which is described in *Remote execution with actions*. It provides a gentle introduction to the distributed aspects of *HPX*.

Tip: Most build targets in *HPX* have two names: a simple name and a hierarchical name corresponding to what type of example or test the target is. If you are developing *HPX* it is often helpful to run `make help` to get a list of available targets. For example, `make help | grep hello_world` outputs the following:

```
... examples.quickstart.hello_world_2
... hello_world_2
... examples.quickstart.hello_world_1
... hello_world_1
... examples.quickstart.hello_world_distributed
... hello_world_distributed
```

It is also possible to build, for instance, all quickstart examples using `make examples.quickstart`.

2.3.7 Creating *HPX* projects

Using HPX with pkg-config

How to build *HPX* applications with pkg-config

After you are done installing *HPX*, you should be able to build the following program. It prints `Hello World!` on the *locality* you run it on.

```
// Including 'hpx/hpx_main.hpp' instead of the usual 'hpx/hpx_init.hpp' enables
// to use the plain C-main below as the direct main HPX entry point.
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

int main()
{
    // Say hello to the world!
    hpx::cout << "Hello World!\n" << std::flush;
    return 0;
}
```

Copy the text of this program into a file called hello_world.cpp.

Now, in the directory where you put hello_world.cpp, issue the following commands (where `$HPX_LOCATION` is the build directory or `CMAKE_INSTALL_PREFIX` you used while building *HPX*):

```
$ export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig
$ c++ -o hello_world hello_world.cpp \
   `pkg-config --cflags --libs hpx_application`\
    -lhpx_iostreams -DHPX_APPLICATION_NAME=hello_world
```

Important: When using pkg-config with *HPX*, the pkg-config flags must go after the `-o` flag.

Note: *HPX* libraries have different names in debug and release mode. If you want to link against a debug *HPX* library, you need to use the `_debug` suffix for the pkg-config name. That means instead of `hpx_application` or `hpx_component`, you will have to use `hpx_application_debug` or `hpx_component_debug` Moreover, all referenced *HPX* components need to have an appended `d` suffix. For example, instead of `-lhpx_iostreams` you will need to specify `-lhpx_iostreamsd`.

Important: If the *HPX* libraries are in a path that is not found by the dynamic linker, you will need to add the path `$HPX_LOCATION/lib` to your linker search path (for example `LD_LIBRARY_PATH` on Linux).

To test the program, type:

```
$ ./hello_world
```

which should print `Hello World!` and exit.

How to build *HPX* components with pkg-config

Let's try a more complex example involving an *HPX* component. An *HPX* component is a class that exposes *HPX* actions. *HPX* components are compiled into dynamically loaded modules called component libraries. Here's the source code:

hello_world_component.cpp

```
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/iostream.hpp>
#include "hello_world_component.hpp"

#include <iostream>

namespace examples { namespace server {
    void hello_world::invoke()
    {
        hpx::cout << "Hello HPX World!" << std::endl;
    }
}}    // namespace examples::server

HPX_REGISTER_COMPONENT_MODULE()

typedef hpx::components::component<examples::server::hello_world>
    hello_world_type;

HPX_REGISTER_COMPONENT(hello_world_type, hello_world)

HPX_REGISTER_ACTION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action)
#endif
```

hello_world_component.hpp

```
#pragma once

#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/hpx.hpp>
#include <hpx/include/actions.hpp>
#include <hpx/include/components.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/serialization.hpp>

#include <utility>

namespace examples { namespace server {
    struct HPX_COMPONENT_EXPORT hello_world
      : hpx::components::component_base<hello_world>
    {
        void invoke();
        HPX_DEFINE_COMPONENT_ACTION(hello_world, invoke)
    };
```

(continues on next page)

(continued from previous page)

```
}}    // namespace examples::server

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action)

namespace examples {
    struct hello_world
      : hpx::components::client_base<hello_world, server::hello_world>
    {
        typedef hpx::components::client_base<hello_world, server::hello_world>
            base_type;

        hello_world(hpx::future<hpx::id_type>&& f)
          : base_type(std::move(f))
        {
        }

        hello_world(hpx::id_type&& f)
          : base_type(std::move(f))
        {
        }

        void invoke()
        {
            hpx::async<server::hello_world::invoke_action>(this->get_id())
                .get();
        }
    };
}    // namespace examples

#endif
```

hello_world_client.cpp

```
#include <hpx/config.hpp>
#if defined(HPX_COMPUTE_HOST_CODE)
#include <hpx/wrap_main.hpp>

#include "hello_world_component.hpp"

int main()
{
    {
        // Create a single instance of the component on this locality.
        examples::hello_world client =
            hpx::new_<examples::hello_world>(hpx::find_here());

        // Invoke the component's action, which will print "Hello World!".
        client.invoke();
    }

    return 0;
```

(continues on next page)

(continued from previous page)

```
}
#endif
```

Copy the three source files above into three files (called `hello_world_component.cpp`, `hello_world_component.hpp` and `hello_world_client.cpp`, respectively).

Now, in the directory where you put the files, run the following command to build the component library. (where `$HPX_LOCATION` is the build directory or `CMAKE_INSTALL_PREFIX` you used while building *HPX*):

```
$ export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig
$ c++ -o libhpx_hello_world.so hello_world_component.cpp \
   `pkg-config --cflags --libs hpx_component` \
    -lhpx_iostreams -DHPX_COMPONENT_NAME=hpx_hello_world
```

Now pick a directory in which to install your *HPX* component libraries. For this example, we'll choose a directory named `my_hpx_libs`:

```
$ mkdir ~/my_hpx_libs
$ mv libhpx_hello_world.so ~/my_hpx_libs
```

Note: *HPX* libraries have different names in debug and release mode. If you want to link against a debug *HPX* library, you need to use the `_debug` suffix for the pkg-config name. That means instead of `hpx_application` or `hpx_component` you will have to use `hpx_application_debug` or `hpx_component_debug`. Moreover, all referenced *HPX* components need to have a appended `d` suffix, e.g. instead of `-lhpx_iostreams` you will need to specify `-lhpx_iostreamsd`.

Important: If the *HPX* libraries are in a path that is not found by the dynamic linker. You need to add the path `$HPX_LOCATION/lib` to your linker search path (for example `LD_LIBRARY_PATH` on Linux).

Now, to build the application that uses this component (`hello_world_client.cpp`), we do:

```
$ export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig
$ c++ -o hello_world_client hello_world_client.cpp \
   ``pkg-config --cflags --libs hpx_application``\
    -L${HOME}/my_hpx_libs -lhpx_hello_world -lhpx_iostreams
```

Important: When using pkg-config with *HPX*, the pkg-config flags must go after the `-o` flag.

Finally, you'll need to set your LD_LIBRARY_PATH before you can run the program. To run the program, type:

```
$ export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$HOME/my_hpx_libs"
$ ./hello_world_client
```

which should print `Hello HPX World!` and exit.

Using *HPX* with CMake-based projects

In addition to the pkg-config support discussed on the previous pages, *HPX* comes with full CMake support. In order to integrate *HPX* into existing or new CMakeLists.txt, you can leverage the find_package[49] command integrated into CMake. Following, is a Hello World component example using CMake.

Let's revisit what we have. We have three files that compose our example application:

- `hello_world_component.hpp`
- `hello_world_component.cpp`
- `hello_world_client.hpp`

The basic structure to include *HPX* into your CMakeLists.txt is shown here:

```
# Require a recent version of cmake
cmake_minimum_required(VERSION 3.18 FATAL_ERROR)

# This project is C++ based.
project(your_app CXX)

# Instruct cmake to find the HPX settings
find_package(HPX)
```

In order to have CMake find *HPX*, it needs to be told where to look for the `HPXConfig.cmake` file that is generated when *HPX* is built or installed. It is used by `find_package(HPX)` to set up all the necessary macros needed to use *HPX* in your project. The ways to achieve this are:

- Set the `HPX_DIR` CMake variable to point to the directory containing the `HPXConfig.cmake` script on the command line when you invoke CMake:

  ```
  $ cmake -DHPX_DIR=$HPX_LOCATION/lib/cmake/HPX ...
  ```

  where `$HPX_LOCATION` is the build directory or `CMAKE_INSTALL_PREFIX` you used when building/configuring *HPX*.

- Set the `CMAKE_PREFIX_PATH` variable to the root directory of your *HPX* build or install location on the command line when you invoke CMake:

  ```
  $ cmake -DCMAKE_PREFIX_PATH=$HPX_LOCATION ...
  ```

  The difference between `CMAKE_PREFIX_PATH` and `HPX_DIR` is that CMake will add common postfixes, such as `lib/cmake/<project`, to the `CMAKE_PREFIX_PATH` and search in these locations too. Note that if your project uses *HPX* as well as other CMake-managed projects, the paths to the locations of these multiple projects may be concatenated in the `CMAKE_PREFIX_PATH`.

- The variables above may be set in the CMake GUI or curses ccmake interface instead of the command line.

Additionally, if you wish to require *HPX* for your project, replace the `find_package(HPX)` line with `find_package(HPX REQUIRED)`.

You can check if *HPX* was successfully found with the `HPX_FOUND` CMake variable.

[49] https://www.cmake.org/cmake/help/latest/command/find_package.html

Using CMake targets

The recommended way of setting up your targets to use *HPX* is to link to the `HPX::hpx` CMake[50] target:

```
target_link_libraries(hello_world_component PUBLIC HPX::hpx)
```

This requires that you have already created the target like this:

```
add_library(hello_world_component SHARED hello_world_component.cpp)
target_include_directories(hello_world_component PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

When you link your library to the `HPX::hpx` CMake[51] target, you will be able use *HPX* functionality in your library. To use `main()` as the implicit entry point in your application you must additionally link your application to the CMake target `HPX::wrap_main`. This target is automatically linked to executables if you are using the macros described below (*Using macros to create new targets*). See *Re-use the main() function as the main HPX entry point* for more information on implicitly using `main()` as the entry point.

Creating a component requires setting two additional compile definitions:

```
target_compile_options(hello_world_component
  HPX_COMPONENT_NAME=hello_world
  HPX_COMPONENT_EXPORTS)
```

Instead of setting these definitions manually you may link to the `HPX::component` target, which sets `HPX_COMPONENT_NAME` to `hpx_<target_name>`, where `<target_name>` is the target name of your library. Note that these definitions should be `PRIVATE` to make sure these definitions are not propagated transitively to dependent targets.

In addition to making your library a component you can make it a plugin. To do so link to the `HPX::plugin` target. Similarly to `HPX::component` this will set `HPX_PLUGIN_NAME` to `hpx_<target_name>`. This definition should also be `PRIVATE`. Unlike regular shared libraries, plugins are loaded at runtime from certain directories and will not be found without additional configuration. Plugins should be installed into a directory containing only plugins. For example, the plugins created by *HPX* itself are installed into the `hpx` subdirectory in the library install directory (typically `lib` or `lib64`). When using the `HPX::plugin` target you need to install your plugins into an appropriate directory. You may also want to set the location of your plugin in the build directory with the `*_OUTPUT_DIRECTORY*` CMake target properties to be able to load the plugins in the build directory. Once you've set the install or output directory of your plugin you need to tell your executable where to find it at runtime. You can do this either by setting the environment variable `HPX_COMPONENT_PATHS` or the ini setting `hpx.component_paths` (see `--hpx:ini`) to the directory containing your plugin.

Using macros to create new targets

In addition to the targets described above, *HPX* provides convenience macros to hide optional boilerplate code that may be useful for your project. The link to the targets described above. We recommend that you use the targets directly whenever possible as they tend to compose better with other targets.

The macro for adding an *HPX* component is `add_hpx_component`. It can be used in your `CMakeLists.txt` file like this:

```
# build your application using HPX
add_hpx_component(hello_world
    SOURCES hello_world_component.cpp
```

(continues on next page)

[50] https://www.cmake.org
[51] https://www.cmake.org

(continued from previous page)

```
    HEADERS hello_world_component.hpp
    COMPONENT_DEPENDENCIES iostreams)
```

Note: add_hpx_component adds a _component suffix to the target name. In the example above, a hello_world_component target will be created.

The available options to add_hpx_component are:

- SOURCES: The source files for that component
- HEADERS: The header files for that component
- DEPENDENCIES: Other libraries or targets this component depends on
- COMPONENT_DEPENDENCIES: The components this component depends on
- PLUGIN: Treats this component as a plugin-able library
- COMPILE_FLAGS: Additional compiler flags
- LINK_FLAGS: Additional linker flags
- FOLDER: Adds the headers and source files to this Source Group folder
- EXCLUDE_FROM_ALL: Do not build this component as part of the all target

After adding the component, the way you add the executable is as follows:

```
# build your application using HPX
add_hpx_executable(hello_world
    SOURCES hello_world_client.cpp
    COMPONENT_DEPENDENCIES hello_world)
```

Note: add_hpx_executable automatically adds a _component suffix to dependencies specified in COMPONENT_DEPENDENCIES, meaning you can directly use the name given when adding a component using add_hpx_component.

When you configure your application, all you need to do is set the HPX_DIR variable to point to the installation of *HPX*.

Note: All library targets built with *HPX* are exported and readily available to be used as arguments to target_link_libraries[52] in your targets. The *HPX* include directories are available with the HPX_INCLUDE_DIRS CMake variable.

[52] https://www.cmake.org/cmake/help/latest/command/target_link_libraries.html

Using the *HPX* compiler wrapper hpxcxx

The hpxcxx compiler wrapper helps to compile a *HPX* component, application, or object file, based on the arguments passed to it.

```
$ hpxcxx [--exe=<APPLICATION_NAME> | --comp=<COMPONENT_NAME> | -c] FLAGS FILES
```

The hpxcxx command **requires** that either an application or a component is built or -c flag is specified. If the build is against a debug build, the -g is to be specified while building.

Optional FLAGS

- -l <LIBRARY> | -l<LIBRARY>: Links <LIBRARY> to the build
- -g: Specifies that the application or component build is against a debug build
- -rd: Sets release-with-debug-info option
- -mr: Sets minsize-release option

All other flags (like -o OUTPUT_FILE) are directly passed to the underlying C++ compiler.

Using macros to set up existing targets to use *HPX*

In addition to the add_hpx_component and add_hpx_executable, you can use the hpx_setup_target macro to have an already existing target to be used with the *HPX* libraries:

```
hpx_setup_target(target)
```

Optional parameters are:

- EXPORT: Adds it to the CMake export list HPXTargets
- INSTALL: Generates an install rule for the target
- PLUGIN: Treats this component as a plugin-able library
- TYPE: The type can be: EXECUTABLE, LIBRARY or COMPONENT
- DEPENDENCIES: Other libraries or targets this component depends on
- COMPONENT_DEPENDENCIES: The components this component depends on
- COMPILE_FLAGS: Additional compiler flags
- LINK_FLAGS: Additional linker flags

If you do not use CMake, you can still build against *HPX*, but you should refer to the section on *How to build HPX components with pkg-config*.

Note: Since *HPX* relies on dynamic libraries, the dynamic linker needs to know where to look for them. If *HPX* isn't installed into a path that is configured as a linker search path, external projects need to either set RPATH or adapt LD_LIBRARY_PATH to point to where the *HPX* libraries reside. In order to set RPATHs, you can include HPX_SetFullRPATH in your project after all libraries you want to link against have been added. Please also consult the CMake documentation here[53].

[53] https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/RPATH-handling

Using *HPX* with Makefile

A basic project building with *HPX* is through creating makefiles. The process of creating one can get complex depending upon the use of cmake parameter HPX_WITH_HPX_MAIN (which defaults to ON).

How to build *HPX* applications with makefile

If *HPX* is installed correctly, you should be able to build and run a simple Hello World program. It prints Hello World! on the *locality* you run it on.

```
// Including 'hpx/hpx_main.hpp' instead of the usual 'hpx/hpx_init.hpp' enables
// to use the plain C-main below as the direct main HPX entry point.
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

int main()
{
    // Say hello to the world!
    hpx::cout << "Hello World!\n" << std::flush;
    return 0;
}
```

Copy the content of this program into a file called hello_world.cpp.

Now, in the directory where you put hello_world.cpp, create a Makefile. Add the following code:

```
CXX=(CXX)  # Add your favourite compiler here or let makefile choose default.

CXXFLAGS=-O3 -std=c++17

Boost_ROOT=/path/to/boost
Hwloc_ROOT=/path/to/hwloc
Tcmalloc_ROOT=/path/to/tcmalloc
HPX_ROOT=/path/to/hpx

INCLUDE_DIRECTIVES=$(HPX_ROOT)/include $(Boost_ROOT)/include $(Hwloc_ROOT)/include

LIBRARY_DIRECTIVES=-L$(HPX_ROOT)/lib $(HPX_ROOT)/lib/libhpx_init.a $(HPX_ROOT)/lib/
↪libhpx.so $(Boost_ROOT)/lib/libboost_atomic-mt.so $(Boost_ROOT)/lib/libboost_
↪filesystem-mt.so $(Boost_ROOT)/lib/libboost_program_options-mt.so $(Boost_ROOT)/lib/
↪libboost_regex-mt.so $(Boost_ROOT)/lib/libboost_system-mt.so -lpthread $(Tcmalloc_
↪ROOT)/libtcmalloc_minimal.so $(Hwloc_ROOT)/libhwloc.so -ldl -lrt

LINK_FLAGS=$(HPX_ROOT)/lib/libhpx_wrap.a -Wl,-wrap=main  # should be left empty for HPX_
↪WITH_HPX_MAIN=OFF

hello_world: hello_world.o
   $(CXX) $(CXXFLAGS) -o hello_world hello_world.o $(LIBRARY_DIRECTIVES) $(LINK_FLAGS)

hello_world.o:
   $(CXX) $(CXXFLAGS) -c -o hello_world.o hello_world.cpp $(INCLUDE_DIRECTIVES)
```

Important: LINK_FLAGS should be left empty if HPX_WITH_HPX_MAIN is set to OFF. Boost in the above example is build with `--layout=tagged`. Actual Boost flags may vary on your build of Boost.

To build the program, type:

```
$ make
```

A successful build should result in hello_world binary. To test, type:

```
$ ./hello_world
```

How to build *HPX* components with makefile

Let's try a more complex example involving an *HPX* component. An *HPX* component is a class that exposes *HPX* actions. *HPX* components are compiled into dynamically-loaded modules called component libraries. Here's the source code:

hello_world_component.cpp

```
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/iostream.hpp>
#include "hello_world_component.hpp"

#include <iostream>

namespace examples { namespace server {
    void hello_world::invoke()
    {
        hpx::cout << "Hello HPX World!" << std::endl;
    }
}}    // namespace examples::server

HPX_REGISTER_COMPONENT_MODULE()

typedef hpx::components::component<examples::server::hello_world>
    hello_world_type;

HPX_REGISTER_COMPONENT(hello_world_type, hello_world)

HPX_REGISTER_ACTION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action)
#endif
```

hello_world_component.hpp

```
#pragma once

#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/hpx.hpp>
#include <hpx/include/actions.hpp>
#include <hpx/include/components.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <hpx/include/lcos.hpp>
#include <hpx/serialization.hpp>

#include <utility>

namespace examples { namespace server {
    struct HPX_COMPONENT_EXPORT hello_world
      : hpx::components::component_base<hello_world>
    {
        void invoke();
        HPX_DEFINE_COMPONENT_ACTION(hello_world, invoke)
    };
}}    // namespace examples::server

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action)

namespace examples {
    struct hello_world
      : hpx::components::client_base<hello_world, server::hello_world>
    {
        typedef hpx::components::client_base<hello_world, server::hello_world>
            base_type;

        hello_world(hpx::future<hpx::id_type>&& f)
          : base_type(std::move(f))
        {
        }

        hello_world(hpx::id_type&& f)
          : base_type(std::move(f))
        {
        }

        void invoke()
        {
            hpx::async<server::hello_world::invoke_action>(this->get_id())
                .get();
        }
    };
}    // namespace examples

#endif
```

hello_world_client.cpp

```
#include <hpx/config.hpp>
#if defined(HPX_COMPUTE_HOST_CODE)
#include <hpx/wrap_main.hpp>

#include "hello_world_component.hpp"

```

(continues on next page)

(continued from previous page)

```
int main()
{
    {
        // Create a single instance of the component on this locality.
        examples::hello_world client =
            hpx::new_<examples::hello_world>(hpx::find_here());

        // Invoke the component's action, which will print "Hello World!".
        client.invoke();
    }

    return 0;
}
#endif
```

Now, in the directory, create a Makefile. Add the following code:

```
CXX=(CXX)  # Add your favourite compiler here or let makefile choose default.

CXXFLAGS=-O3 -std=c++17

Boost_ROOT=/path/to/boost
Hwloc_ROOT=/path/to/hwloc
Tcmalloc_ROOT=/path/to/tcmalloc
HPX_ROOT=/path/to/hpx

INCLUDE_DIRECTIVES=$(HPX_ROOT)/include $(Boost_ROOT)/include $(Hwloc_ROOT)/include

LIBRARY_DIRECTIVES=-L$(HPX_ROOT)/lib $(HPX_ROOT)/lib/libhpx_init.a $(HPX_ROOT)/lib/
↪libhpx.so $(Boost_ROOT)/lib/libboost_atomic-mt.so $(Boost_ROOT)/lib/libboost_
↪filesystem-mt.so $(Boost_ROOT)/lib/libboost_program_options-mt.so $(Boost_ROOT)/lib/
↪libboost_regex-mt.so $(Boost_ROOT)/lib/libboost_system-mt.so -lpthread $(Tcmalloc_
↪ROOT)/libtcmalloc_minimal.so $(Hwloc_ROOT)/libhwloc.so -ldl -lrt

LINK_FLAGS=$(HPX_ROOT)/lib/libhpx_wrap.a -Wl,-wrap=main  # should be left empty for HPX_
↪WITH_HPX_MAIN=OFF

hello_world_client: libhpx_hello_world hello_world_client.o
  $(CXX) $(CXXFLAGS) -o hello_world_client $(LIBRARY_DIRECTIVES) libhpx_hello_world
↪$(LINK_FLAGS)

hello_world_client.o: hello_world_client.cpp
  $(CXX) $(CXXFLAGS) -o hello_world_client.o hello_world_client.cpp $(INCLUDE_DIRECTIVES)

libhpx_hello_world: hello_world_component.o
  $(CXX) $(CXXFLAGS) -o libhpx_hello_world hello_world_component.o $(LIBRARY_DIRECTIVES)

hello_world_component.o: hello_world_component.cpp
  $(CXX) $(CXXFLAGS) -c -o hello_world_component.o hello_world_component.cpp $(INCLUDE_
↪DIRECTIVES)
```

To build the program, type:

```
$ make
```

A successful build should result in hello_world binary. To test, type:

```
$ ./hello_world
```

Note: Due to high variations in CMake flags and library dependencies, it is recommended to build *HPX* applications and components with pkg-config or CMakeLists.txt. Writing Makefile may result in broken builds if due care is not taken. pkg-config files and CMake systems are configured with CMake build of *HPX*. Hence, they are stable when used together and provide better support overall.

2.3.8 Starting the *HPX* runtime

In order to write an application that uses services from the *HPX* runtime system, you need to initialize the *HPX* library by inserting certain calls into the code of your application. Depending on your use case, this can be done in 3 different ways:

- *Minimally invasive*: Re-use the `main()` function as the main *HPX* entry point.
- *Balanced use case*: Supply your own main *HPX* entry point while blocking the main thread.
- *Most flexibility*: Supply your own main *HPX* entry point while avoiding blocking the main thread.
- *Suspend and resume*: As above but suspend and resume the *HPX* runtime to allow for other runtimes to be used.

Re-use the `main()` function as the main *HPX* entry point

This method is the least intrusive to your code. However, it provides you with the smallest flexibility in terms of initializing the *HPX* runtime system. The following code snippet shows what a minimal *HPX* application using this technique looks like:

```
#include <hpx/hpx_main.hpp>

int main(int argc, char* argv[])
{
    return 0;
}
```

The only change to your code you have to make is to include the file `hpx/hpx_main.hpp`. In this case the function `main()` will be invoked as the first *HPX* thread of the application. The runtime system will be initialized behind the scenes before the function `main()` is executed and will automatically stop after `main()` has returned. For this method to work you must link your application to the CMake[54] target `HPX::wrap_main`. This is done automatically if you are using the provided macros (*Using macros to create new targets*) to set up your application, but must be done explicitly if you are using targets directly (*Using CMake targets*). All *HPX* API functions can be used from within the `main()` function now.

Note: The function `main()` does not need to expect receiving `argc` and `argv` as shown above, but could expose the signature `int main()`. This is consistent with the usually allowed prototypes for the function `main()` in C++ applications.

[54] https://www.cmake.org

All command line arguments specific to *HPX* will still be processed by the *HPX* runtime system as usual. However, those command line options will be removed from the list of values passed to argc/argv of the function main(). The list of values passed to main() will hold only the commandline options that are not recognized by the *HPX* runtime system (see the section *HPX Command Line Options* for more details on what options are recognized by *HPX*).

Note: In this mode all one-letter shortcuts that are normally available on the *HPX* command line are disabled (such as -t or -l see *HPX Command Line Options*). This is done to minimize any possible interaction between the command line options recognized by the *HPX* runtime system and any command line options defined by the application.

The value returned from the function main() as shown above will be returned to the operating system as usual.

Important: To achieve this seamless integration, the header file hpx/hpx_main.hpp defines a macro:

```
#define main hpx_startup::user_main
```

which could result in unexpected behavior.

Important: To achieve this seamless integration, we use different implementations for different operating systems. In case of Linux or macOS, the code present in hpx_wrap.cpp is put into action. We hook into the system function in case of Linux and provide alternate entry point in case of macOS. For other operating systems we rely on a macro:

```
#define main hpx_startup::user_main
```

provided in the header file hpx/hpx_main.hpp. This implementation can result in unexpected behavior.

Caution: We make use of an *override* variable include_libhpx_wrap in the header file hpx/hpx_main.hpp to swiftly choose the function call stack at runtime. Therefore, the header file should *only* be included in the main executable. Including it in the components will result in multiple definition of the variable.

Supply your own main *HPX* entry point while blocking the main thread

With this method you need to provide an explicit main-thread function named hpx_main at global scope. This function will be invoked as the main entry point of your *HPX* application on the console *locality* only (this function will be invoked as the first *HPX* thread of your application). All *HPX* API functions can be used from within this function.

The thread executing the function *hpx::init* will block waiting for the runtime system to exit. The value returned from hpx_main will be returned from *hpx::init* after the runtime system has stopped.

The function *hpx::finalize* has to be called on one of the *HPX* localities in order to signal that all work has been scheduled and the runtime system should be stopped after the scheduled work has been executed.

This method of invoking *HPX* has the advantage of the user being able to decide which version of *hpx::init* to call. This allows to pass additional configuration parameters while initializing the *HPX* runtime system.

```
#include <hpx/hpx_init.hpp>

int hpx_main(int argc, char* argv[])
{
    // Any HPX application logic goes here...
```

(continues on next page)

(continued from previous page)

```
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    // Initialize HPX, run hpx_main as the first HPX thread, and
    // wait for hpx::finalize being called.
    return hpx::init(argc, argv);
}
```

Note: The function `hpx_main` does not need to expect receiving `argc/argv` as shown above, but could expose one of the following signatures:

```
int hpx_main();
int hpx_main(int argc, char* argv[]);
int hpx_main(hpx::program_options::variables_map& vm);
```

This is consistent with (and extends) the usually allowed prototypes for the function `main()` in C++ applications.

The header file to include for this method of using *HPX* is `hpx/hpx_init.hpp`.

There are many additional overloads of *`hpx::init`* available, such as the ability to provide your own entry-point function instead of `hpx_main`. Please refer to the function documentation for more details (see: `hpx/hpx_init.hpp`).

Supply your own main *HPX* entry point while avoiding blocking the main thread

With this method you need to provide an explicit main thread function named `hpx_main` at global scope. This function will be invoked as the main entry point of your *HPX* application on the console *locality* only (this function will be invoked as the first *HPX* thread of your application). All *HPX* API functions can be used from within this function.

The thread executing the function *`hpx::start`* will *not* block waiting for the runtime system to exit, but will return immediately. The function *`hpx::finalize`* has to be called on one of the *HPX* localities in order to signal that all work has been scheduled and the runtime system should be stopped after the scheduled work has been executed.

This method of invoking *HPX* is useful for applications where the main thread is used for special operations, such a GUIs. The function *`hpx::stop`* can be used to wait for the *HPX* runtime system to exit and should at least be used as the last function called in `main()`. The value returned from `hpx_main` will be returned from *`hpx::stop`* after the runtime system has stopped.

```
#include <hpx/hpx_start.hpp>

int hpx_main(int argc, char* argv[])
{
    // Any HPX application logic goes here...
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    // Initialize HPX, run hpx_main.
    hpx::start(argc, argv);
```

(continues on next page)

(continued from previous page)

```
    // ...Execute other code here...

    // Wait for hpx::finalize being called.
    return hpx::stop();
}
```

Note: The function `hpx_main` does not need to expect receiving `argc/argv` as shown above, but could expose one of the following signatures:

```
int hpx_main();
int hpx_main(int argc, char* argv[]);
int hpx_main(hpx::program_options::variables_map& vm);
```

This is consistent with (and extends) the usually allowed prototypes for the function `main()` in C++ applications.

The header file to include for this method of using *HPX* is `hpx/hpx_start.hpp`.

There are many additional overloads of *hpx::start* available, such as the option for users to provide their own entry point function instead of `hpx_main`. Please refer to the function documentation for more details (see: `hpx/hpx_start.hpp`).

Supply your own explicit startup function as the main *HPX* entry point

There is also a way to specify any function (besides `hpx_main`) to be used as the main entry point for your *HPX* application:

```
#include <hpx/hpx_init.hpp>

int application_entry_point(int argc, char* argv[])
{
    // Any HPX application logic goes here...
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    // Initialize HPX, run application_entry_point as the first HPX thread,
    // and wait for hpx::finalize being called.
    return hpx::init(&application_entry_point, argc, argv);
}
```

Note: The function supplied to *hpx::init* must have one of the following prototypes:

int application_entry_point(int argc, char* argv[]); int application_entry_point(hpx::program_options::variables_map& vm);

Note: If `nullptr` is used as the function argument, *HPX* will not run any startup function on this locality.

Suspending and resuming the *HPX* runtime

In some applications it is required to combine *HPX* with other runtimes. To support this use case, *HPX* provides two functions: `hpx::suspend` and `hpx::resume`. `hpx::suspend` is a blocking call which will wait for all scheduled tasks to finish executing and then put the thread pool OS threads to sleep. `hpx::resume` simply wakes up the sleeping threads so that they are ready to accept new work. `hpx::suspend` and `hpx::resume` can be found in the header `hpx/hpx_suspend.hpp`.

```
#include <hpx/hpx_start.hpp>
#include <hpx/hpx_suspend.hpp>

int main(int argc, char* argv[])
{
   // Initialize HPX, don't run hpx_main
    hpx::start(nullptr, argc, argv);

    // Schedule a function on the HPX runtime
    hpx::post(&my_function, ...);

    // Wait for all tasks to finish, and suspend the HPX runtime
    hpx::suspend();

    // Execute non-HPX code here

    // Resume the HPX runtime
    hpx::resume();

    // Schedule more work on the HPX runtime

    // hpx::finalize has to be called from the HPX runtime before hpx::stop
    hpx::post([]() { hpx::finalize(); });
    return hpx::stop();
}
```

Note: `hpx::suspend` does not wait for `hpx::finalize` to be called. Only call `hpx::finalize` when you wish to fully stop the *HPX* runtime.

Warning:

`hpx::suspend` only waits for local tasks, i.e. tasks on the current locality, to finish executing. When using `hpx::suspend` in a multi-locality scenario the user is responsible for ensuring that any work required from other localities has also finished.

HPX also supports suspending individual thread pools and threads. For details on how to do that, see the documentation for `hpx::threads::thread_pool_base`.

Automatically suspending worker threads

The previous method guarantees that the worker threads are suspended when you ask for it and that they stay suspended. An alternative way to achieve the same effect is to tweak how quickly *HPX* suspends its worker threads when they run out of work. The following configuration values make sure that *HPX* idles very quickly:

```
hpx.max_idle_backoff_time = 1000
hpx.max_idle_loop_count = 0
```

They can be set on the command line using `--hpx:ini=hpx.max_idle_backoff_time=1000` and `--hpx:ini=hpx.max_idle_loop_count=0`. See *Launching and configuring HPX applications* for more details on how to set configuration parameters.

After setting idling parameters the previous example could now be written like this instead:

```
#include <hpx/hpx_start.hpp>

int main(int argc, char* argv[])
{

   // Initialize HPX, don't run hpx_main
    hpx::start(nullptr, argc, argv);

    // Schedule some functions on the HPX runtime
    // NOTE: run_as_hpx_thread blocks until completion.
    hpx::run_as_hpx_thread(&my_function, ...);
    hpx::run_as_hpx_thread(&my_other_function, ...);

    // hpx::finalize has to be called from the HPX runtime before hpx::stop
    hpx::post([]() { hpx::finalize(); });
    return hpx::stop();
}
```

In this example each call to `hpx::run_as_hpx_thread` acts as a "parallel region".

Working of `hpx_main.hpp`

In order to initialize *HPX* from `main()`, we make use of linker tricks.

It is implemented differently for different operating systems. The method of implementation is as follows:

- *Linux*: Using linker `--wrap` option.
- *Mac OSX*: Using the linker `-e` option.
- *Windows*: Using `#define main hpx_startup::user_main`

Linux implementation

We make use of the Linux linker `ld`'s `--wrap` option to wrap the `main()` function. This way any calls to `main()` are redirected to our own implementation of main. It is here that we check for the existence of `hpx_main.hpp` by making use of a shadow variable `include_libhpx_wrap`. The value of this variable determines the function stack at runtime.

The implementation can be found in `libhpx_wrap.a`.

Important: It is necessary that `hpx_main.hpp` be not included more than once. Multiple inclusions can result in multiple definition of `include_libhpx_wrap`.

Mac OSX implementation

Here we make use of yet another linker option `-e` to change the entry point to our custom entry function `initialize_main`. We initialize the *HPX* runtime system from this function and call main from the initialized system. We determine the function stack at runtime by making use of the shadow variable `include_libhpx_wrap`.

The implementation can be found in `libhpx_wrap.a`.

Important: It is necessary that `hpx_main.hpp` be not included more than once. Multiple inclusions can result in multiple definition of `include_libhpx_wrap`.

Windows implementation

We make use of a macro `#define main hpx_startup::user_main` to take care of the initializations.

This implementation could result in unexpected behaviors.

2.3.9 Launching and configuring *HPX* applications

Configuring *HPX* applications

All *HPX* applications can be configured using special command line options and/or using special configuration files. This section describes the available options, the configuration file format, and the algorithm used to locate possible predefined configuration files. Additionally, this section describes the defaults assumed if no external configuration information is supplied.

During startup any *HPX* application applies a predefined search pattern to locate one or more configuration files. All found files will be read and merged in the sequence they are found into one single internal database holding all configuration properties. This database is used during the execution of the application to configure different aspects of the runtime system.

In addition to the ini files, any application can supply its own configuration files, which will be merged with the configuration database as well. Moreover, the user can specify additional configuration parameters on the command line when executing an application. The *HPX* runtime system will merge all command line configuration options (see the description of the `--hpx:ini`, `--hpx:config`, and `--hpx:app-config` command line options).

The *HPX* ini file format

All *HPX* applications can be configured using a special file format that is similar to the well-known Windows INI file format[55]. This is a structured text format that allows users to group key/value pairs (properties) into sections. The basic element contained in an ini file is the property. Every property has a name and a value, delimited by an equal sign `'='`. The name appears to the left of the equal sign:

```
name=value
```

The value may contain equal signs as only the first `'='` character is interpreted as the delimiter between `name` and `value`. Whitespace before the name, after the value and immediately before and after the delimiting equal sign is ignored. Whitespace inside the value is retained.

Properties may be grouped into arbitrarily named sections. The section name appears on a line by itself, in square brackets. All properties after the section declaration are associated with that section. There is no explicit "end of section" delimiter; sections end at the next section declaration or the end of the file:

```
[section]
```

In *HPX* sections can be nested. A nested section has a name composed of all section names it is embedded in. The section names are concatenated using a dot `'.'`:

```
[outer_section.inner_section]
```

Here, `inner_section` is logically nested within `outer_section`.

It is possible to use the full section name concatenated with the property name to refer to a particular property. For example, in:

```
[a.b.c]
d = e
```

the property value of `d` can be referred to as `a.b.c.d=e`.

In *HPX* ini files can contain comments. Hash signs `'#'` at the beginning of a line indicate a comment. All characters starting with `'#'` until the end of the line are ignored.

If a property with the same name is reused inside a section, the second occurrence of this property name will override the first occurrence (discard the first value). Duplicate sections simply merge their properties together, as if they occurred contiguously.

In *HPX* ini files a property value `${FOO:default}` will use the environmental variable `FOO` to extract the actual value if it is set and `default` otherwise. No default has to be specified. Therefore, `${FOO}` refers to the environmental variable `FOO`. If `FOO` is not set or empty, the overall expression will evaluate to an empty string. A property value `$[section.key:default]` refers to the value held by the property `section.key` if it exists and `default` otherwise. No default has to be specified. Therefore `$[section.key]` refers to the property `section.key`. If the property `section.key` is not set or empty, the overall expression will evaluate to an empty string.

Note: Any property `$[section.key:default]` is evaluated whenever it is queried and not when the configuration data is initialized. This allows for lazy evaluation and relaxes initialization order of different sections. The only exception are recursive property values, e.g., values referring to the very key they are associated with. Those property values are evaluated at initialization time to avoid infinite recursion.

[55] https://en.wikipedia.org/wiki/INI_file

Built-in default configuration settings

During startup any *HPX* application applies a predefined search pattern to locate one or more configuration files. All found files will be read and merged in the sequence they are found into one single internal data structure holding all configuration properties.

As a first step the internal configuration database is filled with a set of default configuration properties. Those settings are described on a section by section basis below.

Note: You can print the default configuration settings used for an executable by specifying the command line option `--hpx:dump-config`.

The `system` configuration section

```
[system]
pid = <process-id>
prefix = <current prefix path of core HPX library>
executable = <current prefix path of executable>
```

Property	Description
`system.pid`	This is initialized to store the current OS-process id of the application instance.
`system.prefix`	This is initialized to the base directory *HPX* has been loaded from.
`system.executable_prefix`	This is initialized to the base directory the current executable has been loaded from.

The `*HPX* configuration section

```
[hpx]
location = ${HPX_LOCATION:$[system.prefix]}
component_path = $[hpx.location]/lib/hpx:$[system.executable_prefix]/lib/hpx:$[system.
↪executable_prefix]/../lib/hpx
master_ini_path = $[hpx.location]/share/hpx-<version>:$[system.executable_prefix]/share/
↪hpx-<version>:$[system.executable_prefix]/../share/hpx-<version>
ini_path = $[hpx.master_ini_path]/ini
os_threads = 1
cores = all
localities = 1
program_name =
cmd_line =
lock_detection = ${HPX_LOCK_DETECTION:0}
throw_on_held_lock = ${HPX_THROW_ON_HELD_LOCK:1}
minimal_deadlock_detection = <debug>
spinlock_deadlock_detection = <debug>
spinlock_deadlock_detection_limit = ${HPX_SPINLOCK_DEADLOCK_DETECTION_LIMIT:1000000}
max_background_threads = ${HPX_MAX_BACKGROUND_THREADS:$[hpx.os_threads]}
max_idle_loop_count = ${HPX_MAX_IDLE_LOOP_COUNT:<hpx_idle_loop_count_max>}
max_busy_loop_count = ${HPX_MAX_BUSY_LOOP_COUNT:<hpx_busy_loop_count_max>}
max_idle_backoff_time = ${HPX_MAX_IDLE_BACKOFF_TIME:<hpx_idle_backoff_time_max>}
```

(continues on next page)

(continued from previous page)

```
exception_verbosity = ${HPX_EXCEPTION_VERBOSITY:2}
trace_depth = ${HPX_TRACE_DEPTH:20}
handle_signals = ${HPX_HANDLE_SIGNALS:1}

[hpx.stacks]
small_size = ${HPX_SMALL_STACK_SIZE:<hpx_small_stack_size>}
medium_size = ${HPX_MEDIUM_STACK_SIZE:<hpx_medium_stack_size>}
large_size = ${HPX_LARGE_STACK_SIZE:<hpx_large_stack_size>}
huge_size = ${HPX_HUGE_STACK_SIZE:<hpx_huge_stack_size>}
use_guard_pages = ${HPX_THREAD_GUARD_PAGE:1}
```

Property	Description
`hpx.location`	This is initialized to the id of the *locality* this application instance is running on.
`hpx.component_path`	Duplicates are discarded. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or by `';'` (Windows).
`hpx.master_ini_path`	This is initialized to the list of default paths of the main hpx.ini configuration files. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows).
`hpx.ini_path`	This is initialized to the default path where *HPX* will look for more ini configuration files. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows).
`hpx.os_threads`	This setting reflects the number of OS threads used for running *HPX* threads. Defaults to number of detected cores (not hyperthreads/PUs).
`hpx.cores`	This setting reflects the number of cores used for running *HPX* threads. Defaults to number of detected cores (not hyperthreads/PUs).
`hpx.localities`	This setting reflects the number of localities the application is running on. Defaults to `1`.
`hpx.program_name`	This setting reflects the program name of the application instance. Initialized from the command line `argv[0]`.
`hpx.cmd_line`	This setting reflects the actual command line used to launch this application instance.
`hpx.lock_detection`	This setting verifies that no locks are being held while a *HPX* thread is suspended. This setting is applicable only if `HPX_WITH_VERIFY_LOCKS` is set during configuration in CMake.
`hpx.throw_on_held_lock`	This setting causes an exception if during lock detection at least one lock is being held while a *HPX* thread is suspended. This setting is applicable only if `HPX_WITH_VERIFY_LOCKS` is set during configuration in CMake. This setting has no effect if `hpx.lock_detection=0`.
`hpx.minimal_deadlock_detection`	This setting enables support for minimal deadlock detection for *HPX* threads. By default this is set to `1` (for Debug builds) or to `0` (for Release, RelWithDebInfo, RelMinSize builds). This setting is effective only if `HPX_WITH_THREAD_DEADLOCK_DETECTION` is set during configuration in CMake.
`hpx.spinlock_deadlock_detection`	This setting verifies that spinlocks don't spin longer than specified using the `hpx.spinlock_deadlock_detection_limit`. This setting is applicable only if `HPX_WITH_SPINLOCK_DEADLOCK_DETECTION` is set during configuration in CMake. By default this is set to `1` (for Debug builds) or to `0` (for Release, RelWithDebInfo, RelMinSize builds).
`hpx.spinlock_deadlock_detection_limit`	This setting specifies the upper limit of the allowed number of spins that spinlocks are allowed to perform. This setting is applicable only if `HPX_WITH_SPINLOCK_DEADLOCK_DETECTION` is set during configuration in CMake. By default this is set to `1000000`.
`hpx.max_background_threads`	This setting defines the number of threads in the scheduler, which are used to execute background work. By default this is the same as the number of cores used for the scheduler.
`hpx.max_idle_loop_count`	By default this is defined by the preprocessor constant `HPX_IDLE_LOOP_COUNT_MAX`. This is an internal setting that you should change only if you know exactly what you are doing.
`hpx.max_busy_loop_count`	This setting defines the maximum value of the busy-loop counter in the scheduler. By default this is defined by the preprocessor constant `HPX_BUSY_LOOP_COUNT_MAX`. This is an internal setting that you should change only if you know exactly what you are doing.
`hpx.max_idle_backoff_time`	This setting defines the maximum time (in milliseconds) for the scheduler to sleep after being idle for `hpx.max_idle_loop_count` iterations. This setting is applicable only if `HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF` is set during configuration in CMake[56]. By default this is defined by the preprocessor constant `HPX_IDLE_BACKOFF_TIME_MAX`. This is an internal setting that you should change only if you know exactly what you are doing.
`hpx.exception_verbosity`	This setting defines the verbosity of exceptions. Valid values are integers. A setting of `2` or higher prints all available information. A setting of `1` leaves out the build configuration and environment variables. A setting of `0` or lower prints only the description of the thrown exception and the file name, function, and line number where the exception was thrown. The default value is `2` or the value of the environment variable `HPX_EXCEPTION_VERBOSITY`.
`hpx.trace_depth`	This setting defines the number of stack-levels printed in generated stack backtraces. This defaults to `20`, but can be changed using the cmake `HPX_WITH_THREAD_BACKTRACE_DEPTH` configuration setting.
`hpx.handle_signals`	This setting defines whether HPX will register signal handlers that will print the configuration information (stack backtrace, system information, etc.) whenever a signal is raised. The default is `1`. Setting this value to `0` can be useful in cases when generating a core-dump on segmentation faults or similar signals is desired. This setting has no effects on non-Linux platforms.

The hpx.threadpools configuration section

```
[hpx.threadpools]
io_pool_size = ${HPX_NUM_IO_POOL_SIZE:2}
parcel_pool_size = ${HPX_NUM_PARCEL_POOL_SIZE:2}
timer_pool_size = ${HPX_NUM_TIMER_POOL_SIZE:2}
```

Property	Description
`hpx.threadpools.io_pool_size`	The value of this property defines the number of OS threads created for the internal I/O thread pool.
`hpx.threadpools.parcel_pool_size`	The value of this property defines the number of OS threads created for the internal parcel thread pool.
`hpx.threadpools.timer_pool_size`	The value of this property defines the number of OS threads created for the internal timer thread pool.

The hpx.thread_queue configuration section

Important: These are the setting control internal values used by the thread scheduling queues in the *HPX* scheduler. You should not modify these settings unless you know exactly what you are doing.

```
[hpx.thread_queue]
min_tasks_to_steal_pending = ${HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_PENDING:0}
min_tasks_to_steal_staged = ${HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_STAGED:0}
min_add_new_count = ${HPX_THREAD_QUEUE_MIN_ADD_NEW_COUNT:10}
max_add_new_count = ${HPX_THREAD_QUEUE_MAX_ADD_NEW_COUNT:10}
max_delete_count = ${HPX_THREAD_QUEUE_MAX_DELETE_COUNT:1000}
```

Property	Description
`hpx.thread_queue.min_tasks_to_steal_pending`	The value of this property defines the number of pending *HPX* threads that have to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.
`hpx.thread_queue.min_tasks_to_steal_staged`	The value of this property defines the number of staged *HPX* tasks that need to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.
`hpx.thread_queue.min_add_new_count`	The value of this property defines the minimal number of tasks to be converted into *HPX* threads whenever the thread queues for a core have run empty.
`hpx.thread_queue.max_add_new_count`	The value of this property defines the maximal number of tasks to be converted into *HPX* threads whenever the thread queues for a core have run empty.
`hpx.thread_queue.max_delete_count`	The value of this property defines the number of terminated *HPX* threads to discard during each invocation of the corresponding function.

[56] https://www.cmake.org

The hpx.components configuration section

```
[hpx.components]
load_external = ${HPX_LOAD_EXTERNAL_COMPONENTS:1}
```

Property	Description
hpx.components.load_external	This entry defines whether external components will be loaded on this *locality*. This entry is normally set to 1, and usually there is no need to directly change this value. It is automatically set to 0 for a dedicated *AGAS* server *locality*.

Additionally, the section hpx.components will be populated with the information gathered from all found components. The information loaded for each of the components will contain at least the following properties:

```
[hpx.components.<component_instance_name>]
name = <component_name>
path = <full_path_of_the_component_module>
enabled = $[hpx.components.load_external]
```

Property	Description
hpx.components.<component_instance_name>.name	This is the name of a component, usually the same as the second argument to the macro used while registering the component with *HPX_REGISTER_COMPONENT*. Set by the component factory.
hpx.components.<component_instance_name>.path	This is either the full path file name of the component module or the directory the component module is located in. In this case, the component module name will be derived from the property hpx.components.<component_instance_name>.name. Set by the component factory.
hpx.components.<component_instance_name>.enabled	This setting explicitly enables or disables the component. This is an optional property. *HPX* assumes that the component is enabled if it is not defined.

The value for <component_instance_name> is usually the same as for the corresponding name property. However, generally it can be defined to any arbitrary instance name. It is used to distinguish between different ini sections, one for each component.

The hpx.parcel configuration section

```
[hpx.parcel]
address = ${HPX_PARCEL_SERVER_ADDRESS:<hpx_initial_ip_address>}
port = ${HPX_PARCEL_SERVER_PORT:<hpx_initial_ip_port>}
bootstrap = ${HPX_PARCEL_BOOTSTRAP:<hpx_parcel_bootstrap>}
max_connections = ${HPX_PARCEL_MAX_CONNECTIONS:<hpx_parcel_max_connections>}
max_connections_per_locality = ${HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY:<hpx_parcel_max_
↪connections_per_locality>}
max_message_size = ${HPX_PARCEL_MAX_MESSAGE_SIZE:<hpx_parcel_max_message_size>}
max_outbound_message_size = ${HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE:<hpx_parcel_max_
↪outbound_message_size>}
array_optimization = ${HPX_PARCEL_ARRAY_OPTIMIZATION:1}
zero_copy_optimization = ${HPX_PARCEL_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.array_
↪optimization]}
zero_copy_receive_optimization = ${HPX_PARCEL_ZERO_COPY_RECEIVE_OPTIMIZATION:$[hpx.
↪parcel.array_optimization]}
```

(continues on next page)

(continued from previous page)

```
async_serialization = ${HPX_PARCEL_ASYNC_SERIALIZATION:1}
message_handlers = ${HPX_PARCEL_MESSAGE_HANDLERS:0}
```

Property	Description
hpx.parcel.address	This property defines the default IP address to be used for the *parcel* layer to listen to. This IP address will be used as long as no other values are specified (for instance, using the *--hpx:hpx* command line option). The expected format is any valid IP address or domain name format that can be resolved into an IP address. The default depends on the compile time preprocessor constant HPX_INITIAL_IP_ADDRESS ("127.0.0.1").
hpx.parcel.port	This property defines the default IP port to be used for the *parcel* layer to listen to. This IP port will be used as long as no other values are specified (for instance using the *--hpx:hpx* command line option). The default depends on the compile time preprocessor constant HPX_INITIAL_IP_PORT (7910).
hpx.parcel.bootstrap	This property defines which parcelport type should be used during application bootstrap. The default depends on the compile time preprocessor constant HPX_PARCEL_BOOTSTRAP ("tcp").
hpx.parcel.max_connections	This property defines how many network connections between different localities are overall kept alive by each *locality*. The default depends on the compile time preprocessor constant HPX_PARCEL_MAX_CONNECTIONS (512).
hpx.parcel.max_connections_per_locality	This property defines the maximum number of network connections that one *locality* will open to another *locality*. The default depends on the compile time preprocessor constant HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY (4).
hpx.parcel.max_message_size	This property defines the maximum allowed message size that will be transferrable through the *parcel* layer. The default depends on the compile time preprocessor constant HPX_PARCEL_MAX_MESSAGE_SIZE (1000000000 bytes).
hpx.parcel.max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size that will be transferrable through the parcel layer. The default depends on the compile time preprocessor constant HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE (1000000 bytes).
hpx.parcel.array_optimization	This property defines whether this *locality* is allowed to utilize array optimizations during serialization of *parcel* data. The default is 1.
hpx.parcel.zero_copy_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations during serialization of *parcel* data. The default is the same value as set for hpx.parcel.array_optimization.
hpx.parcel.zero_copy_receive_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations on the receiving end during de-serialization of *parcel* data. The default is the same value as set for hpx.parcel.zero_copy_optimization.
hpx.parcel.zero_copy_serialization_threshold	This property defines the threshold value (in bytes) starting at which the serialization layer will apply zero-copy optimizations for serialized entities. The default value is defined by the preprocessor constant HPX_ZERO_COPY_SERIALIZATION_THRESHOLD.
hpx.parcel.async_serialization	This property defines whether this *locality* is allowed to spawn a new thread for serialization (this is both for encoding and decoding parcels). The default is 1.
hpx.parcel.message_handlers	This property defines whether message handlers are loaded. The default is 0.
hpx.parcel.max_background_threads	This property defines how many cores should be used to perform background operations. The default is -1 (all cores).

The following settings relate to the TCP/IP parcelport.

```
[hpx.parcel.tcp]
enable = ${HPX_HAVE_PARCELPORT_TCP:$[hpx.parcel.enabled]}
array_optimization = ${HPX_PARCEL_TCP_ARRAY_OPTIMIZATION:$[hpx.parcel.array_
↪optimization]}
zero_copy_optimization = ${HPX_PARCEL_TCP_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.zero_copy_
↪optimization]}
zero_copy_receive_optimization = ${HPX_PARCEL_TCP_ZERO_COPY_RECEIVE_OPTIMIZATION:$[hpx.
↪parcel.zero_copy_receive_optimization]}
zero_copy_serialization_threshold =  ${HPX_PARCEL_TCP_ZERO_COPY_SERIALIZATION_THRESHOLD:
↪$[hpx.parcel.zero_copy_serialization_threshold]}
async_serialization = ${HPX_PARCEL_TCP_ASYNC_SERIALIZATION:$[hpx.parcel.async_
↪serialization]}
parcel_pool_size = ${HPX_PARCEL_TCP_PARCEL_POOL_SIZE:$[hpx.threadpools.parcel_pool_size]}
max_connections =  ${HPX_PARCEL_TCP_MAX_CONNECTIONS:$[hpx.parcel.max_connections]}
max_connections_per_locality = ${HPX_PARCEL_TCP_MAX_CONNECTIONS_PER_LOCALITY:$[hpx.
↪parcel.max_connections_per_locality]}
max_message_size =  ${HPX_PARCEL_TCP_MAX_MESSAGE_SIZE:$[hpx.parcel.max_message_size]}
max_outbound_message_size =  ${HPX_PARCEL_TCP_MAX_OUTBOUND_MESSAGE_SIZE:$[hpx.parcel.max_
↪outbound_message_size]}
max_background_threads =  ${HPX_PARCEL_TCP_MAX_BACKGROUND_THREADS:$[hpx.parcel.max_
↪background_threads]}
```

Property	Description
hpx.parcel.tcp.enable	Enables the use of the default TCP parcelport. Note that the initial bootstrap of the overall *HPX* application will be performed using the default TCP connections. This parcelport is enabled by default. This will be disabled only if MPI is enabled (see below).
hpx.parcel.tcp.array_optimization	This property defines whether this *locality* is allowed to utilize array optimizations in the TCP/IP parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.array_optimization`.
hpx.parcel.tcp.zero_copy_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations during serialization of parcel data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`.
hpx.parcel.tcp.zero_copy_receive_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations on the receiving end in the TCP/IP parcelport during de-serialization of *parcel* data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`.
hpx.parcel.tcp.zero_copy_serialization_threshold	This property defines the threshold value (in bytes) starting at which the serialization layer will apply zero-copy optimizations for serialized entities. The default is the same value as set for `hpx.parcel.zero_copy_serialization_threshold`.
hpx.parcel.tcp.async_serialization	This property defines whether this *locality* is allowed to spawn a new thread for serialization in the TCP/IP parcelport (this is both for encoding and decoding parcels). The default is the same value as set for `hpx.parcel.async_serialization`.
hpx.parcel.tcp.parcel_pool_size	The value of this property defines the number of OS threads created for the internal parcel thread pool of the TCP *parcel* port. The default is taken from `hpx.threadpools.parcel_pool_size`.
hpx.parcel.tcp.max_connections	This property defines how many network connections between different localities are overall kept alive by each *locality*. The default is taken from `hpx.parcel.max_connections`.
hpx.parcel.tcp.max_connections_per_locality	This property defines the maximum number of network connections that one *locality* will open to another *locality*. The default is taken from `hpx.parcel.max_connections_per_locality`.
hpx.parcel.tcp.max_message_size	This property defines the maximum allowed message size that will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_message_size`.
hpx.parcel.tcp.max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size that will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_outbound_connections`.
hpx.parcel.tcp.max_background_threads	This property defines how many cores should be used to perform background operations. The default is taken from `hpx.parcel.max_background_threads`.

The following settings relate to the MPI parcelport. These settings take effect only if the compile time constant `HPX_HAVE_PARCELPORT_MPI` is set (the equivalent CMake variable is `HPX_WITH_PARCELPORT_MPI` and has to be set to `ON`).

```
[hpx.parcel.mpi]
enable = ${HPX_HAVE_PARCELPORT_MPI:$[hpx.parcel.enabled]}
env = ${HPX_HAVE_PARCELPORT_MPI_ENV:MV2_COMM_WORLD_RANK,PMI_RANK,OMPI_COMM_WORLD_SIZE,
↪ALPS_APP_PE,PALS_NODEID}
multithreaded = ${HPX_HAVE_PARCELPORT_MPI_MULTITHREADED:1}
rank = <MPI_rank>
processor_name = <MPI_processor_name>
array_optimization = ${HPX_HAVE_PARCEL_MPI_ARRAY_OPTIMIZATION:$[hpx.parcel.array_
↪optimization]}
zero_copy_optimization = ${HPX_HAVE_PARCEL_MPI_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.zero_
↪copy_optimization]}
zero_copy_receive_optimization = ${HPX_HAVE_PARCEL_MPI_ZERO_COPY_RECEIVE_OPTIMIZATION:
↪$[hpx.parcel.zero_copy_receive_optimization]}
```

(continues on next page)

(continued from previous page)

```
zero_copy_serialization_threshold =  ${HPX_PARCEL_MPI_ZERO_COPY_SERIALIZATION_THRESHOLD:
↪$[hpx.parcel.zero_copy_serialization_threshold]}
use_io_pool = ${HPX_HAVE_PARCEL_MPI_USE_IO_POOL:$1}
async_serialization = ${HPX_HAVE_PARCEL_MPI_ASYNC_SERIALIZATION:$[hpx.parcel.async_
↪serialization]}
parcel_pool_size = ${HPX_HAVE_PARCEL_MPI_PARCEL_POOL_SIZE:$[hpx.threadpools.parcel_pool_
↪size]}
max_connections =  ${HPX_HAVE_PARCEL_MPI_MAX_CONNECTIONS:$[hpx.parcel.max_connections]}
max_connections_per_locality = ${HPX_HAVE_PARCEL_MPI_MAX_CONNECTIONS_PER_LOCALITY:$[hpx.
↪parcel.max_connections_per_locality]}
max_message_size =  ${HPX_HAVE_PARCEL_MPI_MAX_MESSAGE_SIZE:$[hpx.parcel.max_message_
↪size]}
max_outbound_message_size =  ${HPX_HAVE_PARCEL_MPI_MAX_OUTBOUND_MESSAGE_SIZE:$[hpx.
↪parcel.max_outbound_message_size]}
max_background_threads =  ${HPX_PARCEL_MPI_MAX_BACKGROUND_THREADS:$[hpx.parcel.max_
↪background_threads]}
```

Property	Description
hpx.parcel.mpi.enable	Enables the use of the MPI parcelport. *HPX* tries to detect if the application was started within a parallel MPI environment. If the detection was successful, the MPI parcelport is enabled by default. To explicitly disable the MPI parcelport, set to 0. Note that the initial bootstrap of the overall *HPX* application will be performed using MPI as well.
hpx.parcel.mpi.env	This property influences which environment variables (separated by commas) will be analyzed to find out whether the application was invoked by MPI.
hpx.parcel.mpi.multithreaded	This property is used to determine what threading mode to use when initializing MPI. If this setting is `0`, *HPX* will initialize MPI with `MPI_THREAD_SINGLE`. If the value is not equal to `0`, *HPX* will initialize MPI with `MPI_THREAD_MULTI`.
hpx.parcel.mpi.rank	This property will be initialized to the MPI rank of the *locality*.
hpx.parcel.mpi.processor_name	This property will be initialized to the MPI processor name of the *locality*.
hpx.parcel.mpi.array_optimization	This property defines whether this *locality* is allowed to utilize array optimizations in the MPI parcelport during serialization of *parcel* data. The default is the same value as set for `hpx.parcel.array_optimization`.
hpx.parcel.mpi.zero_copy_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations in the MPI parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`.
hpx.parcel.mpi.zero_copy_receive_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations on the receiving end in the MPI parcelport during de-serialization of *parcel* data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`.
hpx.parcel.mpi.zero_copy_serialization_threshold	This property defines the threshold value (in bytes) starting at which the serialization layer will apply zero-copy optimizations for serialized entities. The default is the same value as set for `hpx.parcel.zero_copy_serialization_threshold`.
hpx.parcel.mpi.use_io_pool	This property can be set to run the progress thread inside of *HPX* threads instead of a separate thread pool. The default is 1.
hpx.parcel.mpi.async_serialization	This property defines whether this *locality* is allowed to spawn a new thread for serialization in the MPI parcelport (this is both for encoding and decoding parcels). The default is the same value as set for `hpx.parcel.async_serialization`.
hpx.parcel.mpi.parcel_pool_size	The value of this property defines the number of OS threads created for the internal parcel thread pool of the MPI *parcel* port. The default is taken from `hpx.threadpools.parcel_pool_size`.
hpx.parcel.mpi.max_connections	This property defines how many network connections between different localities are overall kept alive by each *locality*. The default is taken from `hpx.parcel.max_connections`.
hpx.parcel.mpi.max_connections_per_locality	This property defines the maximum number of network connections that one *locality* will open to another *locality*. The default is taken from `hpx.parcel.max_connections_per_locality`.
hpx.parcel.mpi.max_message_size	This property defines the maximum allowed message size that will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_message_size`.
hpx.parcel.mpi.max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size that will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_outbound_connections`.
hpx.parcel.mpi.max_background_threads	This property defines how many cores should be used to perform background operations. The default is taken from `hpx.parcel.max_background_threads`.

The hpx.agas configuration section

```
[hpx.agas]
address = ${HPX_AGAS_SERVER_ADDRESS:<hpx_initial_ip_address>}
port = ${HPX_AGAS_SERVER_PORT:<hpx_initial_ip_port>}
service_mode = hosted
dedicated_server = 0
max_pending_refcnt_requests = ${HPX_AGAS_MAX_PENDING_REFCNT_REQUESTS:<hpx_initial_agas_
↪max_pending_refcnt_requests>}
use_caching = ${HPX_AGAS_USE_CACHING:1}
use_range_caching = ${HPX_AGAS_USE_RANGE_CACHING:1}
local_cache_size = ${HPX_AGAS_LOCAL_CACHE_SIZE:<hpx_agas_local_cache_size>}
```

Property	Description
`hpx.agas.address`	This property defines the default IP address to be used for the *AGAS* root server. This IP address will be used as long as no other values are specified (for instance, using the *--hpx:agas* command line option). The expected format is any valid IP address or domain name format that can be resolved into an IP address. The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_ADDRESS` ("127.0.0.1").
`hpx.agas.port`	This property defines the default IP port to be used for the *AGAS* root server. This IP port will be used as long as no other values are specified (for instance, using the *--hpx:agas* command line option). The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_PORT` (7009).
`hpx.agas.service_mode`	This property specifies what type of *AGAS* service is running on this *locality*. Currently, two modes exist. The *locality* that acts as the *AGAS* server runs in `bootstrap` mode. All other localities are in `hosted` mode.
`hpx.agas.dedicated_server`	This property specifies whether the *AGAS* server is exclusively running *AGAS* services and not hosting any application components. It is a boolean value. Set to 1 if *--hpx:run-agas-server-only* is present.
`hpx.agas.max_pending_refcnt_requests`	This property defines the number of reference counting requests (increments or decrements) to buffer. The default depends on the compile time preprocessor constant `HPX_INITIAL_AGAS_MAX_PENDING_REFCNT_REQUESTS` (4096).
`hpx.agas.use_caching`	This property specifies whether a software address translation cache is used. It is a boolean value. Defaults to 1.
`hpx.agas.use_range_caching`	This property specifies whether range-based caching is used by the software address translation cache. This property is ignored if *hpx.agas.use_caching* is false. It is a boolean value. Defaults to 1.
`hpx.agas.local_cache_size`	This property defines the size of the software address translation cache for *AGAS* services. This property is ignored if `hpx.agas.use_caching` is false. Note that if `hpx.agas.use_range_caching` is true, this size will refer to the maximum number of ranges stored in the cache, not the number of entries spanned by the cache. The default depends on the compile time preprocessor constant `HPX_AGAS_LOCAL_CACHE_SIZE` (4096).

The hpx.commandline configuration section

The following table lists the definition of all pre-defined command line option shortcuts. For more information about commandline options, see the section *HPX Command Line Options*.

```
[hpx.commandline]
aliasing = ${HPX_COMMANDLINE_ALIASING:1}
allow_unknown = ${HPX_COMMANDLINE_ALLOW_UNKNOWN:0}

[hpx.commandline.aliases]
-a = --hpx:agas
-c = --hpx:console
-h = --hpx:help
-I = --hpx:ini
-l = --hpx:localities
-p = --hpx:app-config
-q = --hpx:queuing
-r = --hpx:run-agas-server
-t = --hpx:threads
-v = --hpx:version
-w = --hpx:worker
-x = --hpx:hpx
-0 = --hpx:node=0
-1 = --hpx:node=1
-2 = --hpx:node=2
-3 = --hpx:node=3
-4 = --hpx:node=4
-5 = --hpx:node=5
-6 = --hpx:node=6
-7 = --hpx:node=7
-8 = --hpx:node=8
-9 = --hpx:node=9
```

Note: The short options listed above are disabled by default if the application is built using `#include <hpx/hpx_main.hpp>`. See *Re-use the main() function as the main HPX entry point* for more information. The rationale behind this is that in this case the user's application may handle its own command line options, since *HPX* passes all unknown options to `main()`. Short options like `-t` are prone to create ambiguities regarding what the application will support. Hence, the user should instead rely on the corresponding long options like `--hpx:threads` in such a case.

Property	Description
hpx.commandline.aliasing	Enable command line aliases as defined in the section hpx.commandline.aliases (see below). Defaults to 1.
hpx.commandline.allow_unknown	Allow for unknown command line options to be passed through to hpx_main() Defaults to 0.
hpx.commandline.aliases.-a	On the commandline -a expands to: *--hpx:agas*.
hpx.commandline.aliases.-c	On the commandline -c expands to: *--hpx:console*.
hpx.commandline.aliases.-h	On the commandline -h expands to: *--hpx:help*.
hpx.commandline.aliases.--help	On the commandline --help expands to: *--hpx:help*.
hpx.commandline.aliases.-I	On the commandline -I expands to: *--hpx:ini*.
hpx.commandline.aliases.-l	On the commandline -l expands to: *--hpx:localities*.
hpx.commandline.aliases.-p	On the commandline -p expands to: *--hpx:app-config*.
hpx.commandline.aliases.-q	On the commandline -q expands to: *--hpx:queuing*.
hpx.commandline.aliases.-r	On the commandline -r expands to: *--hpx:run-agas-server*.
hpx.commandline.aliases.-t	On the commandline -t expands to: *--hpx:threads*.
hpx.commandline.aliases.-v	On the commandline -v expands to: *--hpx:version*.
hpx.commandline.aliases.--version	On the commandline --version expands to: *--hpx:version*.
hpx.commandline.aliases.-w	On the commandline -w expands to: *--hpx:worker*.
hpx.commandline.aliases.-x	On the commandline -x expands to: *--hpx:hpx*.
hpx.commandline.aliases.-0	On the commandline -0 expands to: *--hpx:node*=0.
hpx.commandline.aliases.-1	On the commandline -1 expands to: *--hpx:node*=1.
hpx.commandline.aliases.-2	On the commandline -2 expands to: *--hpx:node*=2.
hpx.commandline.aliases.-3	On the commandline -3 expands to: *--hpx:node*=3.
hpx.commandline.aliases.-4	On the commandline -4 expands to: *--hpx:node*=4.
hpx.commandline.aliases.-5	On the commandline -5 expands to: *--hpx:node*=5.
hpx.commandline.aliases.-6	On the commandline -6 expands to: *--hpx:node*=6.
hpx.commandline.aliases.-7	On the commandline -7 expands to: *--hpx:node*=7.
hpx.commandline.aliases.-8	On the commandline -8 expands to: *--hpx:node*=8.
hpx.commandline.aliases.-9	On the commandline -9 expands to: *--hpx:node*=9.

Loading INI files

During startup and after the internal database has been initialized as described in the section *Built-in default configuration settings*, *HPX* will try to locate and load additional ini files to be used as a source for configuration properties. This allows for a wide spectrum of additional customization possibilities by the user and system administrators. The sequence of locations where *HPX* will try loading the ini files is well defined and documented in this section. All ini files found are merged into the internal configuration database. The merge operation itself conforms to the rules as described in the section *The HPX ini file format*.

1. Load all component shared libraries found in the directories specified by the property `hpx.component_path` and retrieve their default configuration information (see section *Loading components* for more details). This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or by `';'` (Windows).
2. Load all files named `hpx.ini` in the directories referenced by the property `hpx.master_ini_path` This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or by `';'` (Windows).
3. Load a file named `.hpx.ini` in the current working directory, e.g., the directory the application was invoked from.
4. Load a file referenced by the environment variable `HPX_INI`. This variable is expected to provide the full path name of the ini configuration file (if any).
5. Load a file named `/etc/hpx.ini`. This lookup is done on non-Windows systems only.
6. Load a file named `.hpx.ini` in the home directory of the current user, e.g., the directory referenced by the environment variable `HOME`.
7. Load a file named `.hpx.ini` in the directory referenced by the environment variable `PWD`.
8. Load the file specified on the command line using the option `--hpx:config`.
9. Load all properties specified on the command line using the option `--hpx:ini`. The properties will be added to the database in the same sequence as they are specified on the command line. The format for those options is, for instance, `--hpx:ini=hpx.default_stack_size=0x4000`. In addition to the explicit command line options, this will set the following properties as implied from other settings:
   - `hpx.parcel.address` and `hpx.parcel.port` as set by `--hpx:hpx`
   - `hpx.agas.address`, `hpx.agas.port` and `hpx.agas.service_mode` as set by `--hpx:agas`
   - `hpx.program_name` and `hpx.cmd_line` will be derived from the actual command line
   - **`hpx.os_threads` and `hpx.localities` as set by** `--hpx:threads` and `--hpx:localities`
   - `hpx.runtime_mode` will be derived from any explicit `--hpx:console`, `--hpx:worker`, or `--hpx:connect`, or it will be derived from other settings, such as `--hpx:node =0`, which implies `--hpx:console`.
10. Load files based on the pattern `*.ini` in all directories listed by the property `hpx.ini_path`. All files found during this search will be merged. The property `hpx.ini_path` can hold a list of directories separated by `':'` (on Linux or Mac) or `';'` (on Windows).
11. Load the file specified on the command line using the option `--hpx:app-config`. Note that this file will be merged as the content for a top level section `[application]`.

Note: Any changes made to the configuration database caused by one of the steps will influence the loading process for all subsequent steps. For instance, if one of the ini files loaded changes the property `hpx.ini_path`, this will influence the directories searched in step 9 as described above.

Important: The *HPX* core library will verify that all configuration settings specified on the command line (using the `--hpx:ini` option) will be checked for validity. That means that the library will accept only *known* configuration settings. This is to protect the user from unintentional typos while specifying those settings. This behavior can be overwritten by appending a `'!'` to the configuration key, thus forcing the setting to be entered into the configuration database. For instance: `--hpx:ini=hpx.foo! = 1`

If any of the environment variables or files listed above are not found, the corresponding loading step will be silently skipped.

Loading components

HPX relies on loading application specific components during the runtime of an application. Moreover, *HPX* comes with a set of preinstalled components supporting basic functionalities useful for almost every application. Any component in *HPX* is loaded from a shared library, where any of the shared libraries can contain more than one component type. During startup, *HPX* tries to locate all available components (e.g., their corresponding shared libraries) and creates an internal component registry for later use. This section describes the algorithm used by *HPX* to locate all relevant shared libraries on a system. As described, this algorithm is customizable by the configuration properties loaded from the ini files (see section *Loading INI files*).

Loading components is a two-stage process. First *HPX* tries to locate all component shared libraries, loads those, and generates a default configuration section in the internal configuration database for each component found. For each found component the following information is generated:

```
[hpx.components.<component_instance_name>]
name = <name_of_shared_library>
path = $[component_path]
enabled = $[hpx.components.load_external]
default = 1
```

The values in this section correspond to the expected configuration information for a component as described in the section *Built-in default configuration settings*.

In order to locate component shared libraries, *HPX* will try loading all shared libraries (files with the platform specific extension of a shared library, Linux: `*.so`, Windows: `*.dll`, MacOS: `*.dylib` found in the directory referenced by the ini property `hpx.component_path`).

This first step corresponds to step 1) during the process of filling the internal configuration database with default information as described in section *Loading INI files*.

After all of the configuration information has been loaded, *HPX* performs the second step in terms of loading components. During this step, *HPX* scans all existing configuration sections `[hpx.component.<some_component_instance_name>]` and instantiates a special factory object for each of the successfully located and loaded components. During the application's life time, these factory objects are responsible for creating new and discarding old instances of the component they are associated with. This step is performed after step 11) of the process of filling the internal configuration database with default information as described in section *Loading INI files*.

Application specific component example

This section assumes there is a simple application component that exposes one member function as a component action. The header file `app_server.hpp` declares the C++ type to be exposed as a component. This type has a member function `print_greeting()`, which is exposed as an action `print_greeting_action`. We assume the source files for this example are located in a directory referenced by `$APP_ROOT`:

```
// file: $APP_ROOT/app_server.hpp
#include <hpx/hpx.hpp>
#include <hpx/include/iostreams.hpp>

namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::component_base<server>
    {
        void print_greeting ()
        {
            hpx::cout << "Hey, how are you?\n" << std::flush;
        }

        // Component actions need to be declared, this also defines the
        // type 'print_greeting_action' representing the action.
        HPX_DEFINE_COMPONENT_ACTION(server, print_greeting, print_greeting_action);
    };
}

// Declare boilerplate code required for each of the component actions.
HPX_REGISTER_ACTION_DECLARATION(app::server::print_greeting_action);
```

The corresponding source file contains mainly macro invocations that define the boilerplate code needed for *HPX* to function properly:

```
// file: $APP_ROOT/app_server.cpp
#include "app_server.hpp"

// Define boilerplate required once per component module.
HPX_REGISTER_COMPONENT_MODULE();

// Define factory object associated with our component of type 'app::server'.
HPX_REGISTER_COMPONENT(app::server, app_server);

// Define boilerplate code required for each of the component actions. Use the
// same argument as used for HPX_REGISTER_ACTION_DECLARATION above.
HPX_REGISTER_ACTION(app::server::print_greeting_action);
```

The following gives an example of how the component can be used. Here, one instance of the `app::server` component is created on the current *locality* and the exposed action `print_greeting_action` is invoked using the global id of the newly created instance. Note that no special code is required to delete the component instance after it is not needed anymore. It will be deleted automatically when its last reference goes out of scope (shown in the example below at the closing brace of the block surrounding the code):

```
// file: $APP_ROOT/use_app_server_example.cpp
#include <hpx/hpx_init.hpp>
#include "app_server.hpp"

int hpx_main()
{
    {
        // Create an instance of the app_server component on the current locality.
        hpx::naming:id_type app_server_instance =
            hpx::create_component<app::server>(hpx::find_here());

        // Create an instance of the action 'print_greeting_action'.
        app::server::print_greeting_action print_greeting;

        // Invoke the action 'print_greeting' on the newly created component.
        print_greeting(app_server_instance);
    }
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);
}
```

In order to make sure that the application will be able to use the component `app::server`, special configuration information must be passed to *HPX*. The simplest way to allow *HPX* to 'find' the component is to provide special ini configuration files that add the necessary information to the internal configuration database. The component should have a special ini file containing the information specific to the component `app_server`.

```
# file: $APP_ROOT/app_server.ini
[hpx.components.app_server]
name = app_server
path = $APP_LOCATION/
```

Here, `$APP_LOCATION` is the directory where the (binary) component shared library is located. *HPX* will attempt to load the shared library from there. The section name `hpx.components.app_server` reflects the instance name of the component (`app_server` is an arbitrary, but unique name). The property value for `hpx.components.app_server.name` should be the same as used for the second argument to the macro *HPX_REGISTER_COMPONENT* above.

Additionally, a file `.hpx.ini`, which could be located in the current working directory (see step 3 as described in the section *Loading INI files*), can be used to add to the ini search path for components:

```
# file: $PWD/.hpx.ini
[hpx]
ini_path = $[hpx.ini_path]:$APP_ROOT/
```

This assumes that the above ini file specific to the component is located in the directory `$APP_ROOT`.

Note: It is possible to reference the defined property from inside its value. *HPX* will gracefully use the previous value of `hpx.ini_path` for the reference on the right hand side and assign the overall (now expanded) value to the property.

Logging

HPX uses a sophisticated logging framework, allowing users to follow in detail what operations have been performed inside the *HPX* library in what sequence. This information proves to be very useful for diagnosing problems or just for improving the understanding of what is happening in *HPX* as a consequence of invoking *HPX* API functionality.

Default logging

Enabling default logging is a simple process. The detailed description in the remainder of this section explains different ways to customize the defaults. Default logging can be enabled by using one of the following:

- A command line switch *--hpx:debug-hpx-log*, which will enable logging to the console terminal.
- The command line switch *--hpx:debug-hpx-log*=<filename>, which enables logging to a given file <filename>.
- Setting an environment variable HPX_LOGLEVEL=<loglevel> while running the *HPX* application. In this case <loglevel> should be a number between (or equal to) 1 and 5 where 1 means minimal logging and 5 causes all available messages to be logged. When setting the environment variable, the logs will be written to a file named hpx.<PID>.lo in the current working directory, where <PID> is the process id of the console instance of the application.

Customizing logging

Generally, logging can be customized either using environment variable settings or using by an ini configuration file. Logging is generated in several categories, each of which can be customized independently. All customizable configuration parameters have reasonable defaults, allowing for the use of logging without any additional configuration effort. The following table lists the available categories.

Table 2.5: Logging categories

Category	Category shortcut	Information to be generated	Environment variable
General	None	Logging information generated by different subsystems of *HPX*, such as thread-manager, parcel layer, LCOs, etc.	HPX_LOGLEVEL
AGAS	AGAS	Logging output generated by the *AGAS* subsystem	HPX_AGAS_LOGLEVEL
Application	APP	Logging generated by applications.	HPX_APP_LOGLEVEL

By default, all logging output is redirected to the console instance of an application, where it is collected and written to a file, one file for each logging category.

Each logging category can be customized at two levels. The parameters for each are stored in the ini configuration sections hpx.logging.CATEGORY and hpx.logging.console.CATEGORY (where CATEGORY is the category shortcut as listed in the table above). The former influences logging at the source *locality* and the latter modifies the logging behaviour for each of the categories at the console instance of an application.

Levels

All *HPX* logging output has seven different logging levels. These levels can be set explicitly or through environment variables in the main *HPX* ini file as shown below. The logging levels and their associated integral values are shown in the table below, ordered from most verbose to least verbose. By default, all *HPX* logs are set to 0, e.g., all logging output is disabled by default.

Table 2.6: Logging levels

Logging level	Integral value
<debug>	5
<info>	4
<warning>	3
<error>	2
<fatal>	1
No logging	0

Tip: The easiest way to enable logging output is to set the environment variable corresponding to the logging category to an integral value as described in the table above. For instance, setting HPX_LOGLEVEL=5 will enable full logging output for the general category. Please note that the syntax and means of setting environment variables varies between operating systems.

Configuration

Logs will be saved to destinations as configured by the user. By default, logging output is saved on the console instance of an application to hpx.<CATEGORY>.<PID>.lo (where CATEGORY and PID> are placeholders for the category shortcut and the OS process id). The output for the general logging category is saved to hpx.<PID>.log. The default settings for the general logging category are shown here (the syntax is described in the section *The HPX ini file format*):

```
[hpx.logging]
level = ${HPX_LOGLEVEL:0}
destination = ${HPX_LOGDESTINATION:console}
format = ${HPX_LOGFORMAT:(T%locality%/%hpxthread%.%hpxphase%/%hpxcomponent%) P%parentloc
↪%/%hpxparent%.%hpxparentphase% %time%($hh:$mm.$ss.$mili) [%idx%]|\\n}
```

The logging level is taken from the environment variable HPX_LOGLEVEL and defaults to zero, e.g., no logging. The default logging destination is read from the environment variable HPX_LOGDESTINATION On any of the localities it defaults to console, which redirects all generated logging output to the console instance of an application. The following table lists the possible destinations for any logging output. It is possible to specify more than one destination separated by whitespace.

Table 2.7: Logging destinations

Logging destination	Description
file(<filename>)	Directs all output to a file with the given <filename>.
cout	Directs all output to the local standard output of the application instance on this *locality*.
cerr	Directs all output to the local standard error output of the application instance on this *locality*.
console	Directs all output to the console instance of the application. The console instance has its logging destinations configured separately.
android_log	Directs all output to the (Android) system log (available on Android systems only).

The logging format is read from the environment variable HPX_LOGFORMAT, and it defaults to a complex format description. This format consists of several placeholder fields (for instance %locality%), which will be replaced by concrete values when the logging output is generated. All other information is transferred verbatim to the output. The table below describes the available field placeholders. The separator character | separates the logging message prefix formatted as shown and the actual log message which will replace the separator.

Table 2.8: Available field placeholders

Name	Description
locality	The id of the *locality* on which the logging message was generated.
hpxthread	The id of the *HPX* thread generating this logging output.
hpxphase	The phase[58] of the *HPX* thread generating this logging output.
hpxcomponent	The local virtual address of the component which the current *HPX* thread is accessing.
parentloc	The id of the *locality* where the *HPX* thread was running that initiated the current *HPX* thread. The current *HPX* thread is generating this logging output.
hpxparent	The id of the *HPX* thread that initiated the current *HPX* thread. The current *HPX* thread is generating this logging output.
hpxparentphase	The phase of the *HPX* thread when it initiated the current *HPX* thread. The current *HPX* thread is generating this logging output.
time	The time stamp for this logging outputline as generated by the source *locality*.
idx	The sequence number of the logging output line as generated on the source *locality*.
osthread	The sequence number of the OS thread that executes the current *HPX* thread.

Note: Not all of the field placeholder may be expanded for all generated logging output. If no value is available for a particular field, it is replaced with a sequence of '-' characters.

Here is an example line from a logging output generated by one of the *HPX* examples (please note that this is generated on a single line, without a line break):

```
(T00000000/0000000002d46f90.01/00000000009ebc10) P--------/0000000002d46f80.02 17:49.37.
↪320 [000000000000004d]
    <info>  [RT] successfully created component {0000000100ff0001, 0000000000030002} of␣
↪type: component_barrier[7(3)]
```

The default settings for the general logging category on the console is shown here:

```
[hpx.logging.console]
level = ${HPX_LOGLEVEL:$[hpx.logging.level]}
destination = ${HPX_CONSOLE_LOGDESTINATION:file(hpx.$[system.pid].log)}
format = ${HPX_CONSOLE_LOGFORMAT:|}
```

These settings define how the logging is customized once the logging output is received by the console instance of an application. The logging level is read from the environment variable HPX_LOGLEVEL (as set for the console instance of the application). The level defaults to the same values as the corresponding settings in the general logging configuration shown before. The destination on the console instance is set to be a file that's name is generated based on its OS process id. Setting the environment variable HPX_CONSOLE_LOGDESTINATION allows customization of the naming scheme for the output file. The logging format is set to leave the original logging output unchanged, as received from one of the localities the application runs on.

[58] The phase of a *HPX*-thread counts how often this thread has been activated.

HPX Command Line Options

The predefined command line options for any application using `hpx::init` are described in the following subsections.

HPX options (allowed on command line only)

`--hpx:help`
: Print out program usage (default: this message). Possible values: `full` (additionally prints options from components).

`--hpx:version`
: Print out *HPX* version and copyright information.

`--hpx:info`
: Print out *HPX* configuration information.

`--hpx:options-file` `arg`
: Specify a file containing command line options (alternatively: @filepath).

HPX options (additionally allowed in an options file)

`--hpx:worker`
: Run this instance in worker mode.

`--hpx:console`
: Run this instance in console mode.

`--hpx:connect`
: Run this instance in worker mode, but connecting late.

`--hpx:run-agas-server`
: Run *AGAS* server as part of this runtime instance.

`--hpx:run-hpx-main`
: Run the hpx_main function, regardless of *locality* mode.

`--hpx:hpx` `arg`
: The IP address the *HPX* parcelport is listening on, expected format: `address:port` (default: `127.0.0.1:7910`).

`--hpx:agas` `arg`
: The IP address the *AGAS* root server is running on, expected format: `address:port` (default: `127.0.0.1:7910`).

`--hpx:run-agas-server-only`
: Run only the *AGAS* server.

`--hpx:nodefile` `arg`
: The file name of a node file to use (list of nodes, one node name per line and core).

`--hpx:nodes` `arg`
: The (space separated) list of the nodes to use (usually this is extracted from a node file).

`--hpx:endnodes`
: This can be used to end the list of nodes specified using the option `--hpx:nodes`.

`--hpx:ifsuffix arg`
Suffix to append to host names in order to resolve them to the proper network interconnect.

`--hpx:ifprefix arg`
Prefix to prepend to host names in order to resolve them to the proper network interconnect.

`--hpx:iftransform arg`
Sed-style search and replace (`s/search/replace/`) used to transform host names to the proper network interconnect.

`--hpx:force_ipv4`
Network hostnames will be resolved to ipv4 addresses instead of using the first resolved endpoint. This is especially useful on Windows where the local hostname will resolve to an ipv6 address while remote network hostnames are commonly resolved to ipv4 addresses.

`--hpx:localities arg`
The number of localities to wait for at application startup (default: `1`).

`--hpx:node arg`
Number of the node this *locality* is run on (must be unique).

`--hpx:ignore-batch-env`
Ignore batch environment variables.

`--hpx:expect-connecting-localities`
This *locality* expects other localities to dynamically connect (this is implied if the number of initial localities is larger than 1).

`--hpx:pu-offset`
The first processing unit this instance of *HPX* should be run on (default: `0`).

`--hpx:pu-step`
The step between used processing unit numbers for this instance of *HPX* (default: `1`).

`--hpx:threads arg`
The number of operating system threads to spawn for this *HPX* *locality*. Possible values are: numeric values `1`, `2`, `3` and so on, `all` (which spawns one thread per processing unit, includes hyperthreads), or `cores` (which spawns one thread per core) (default: `cores`).

`--hpx:cores arg`
The number of cores to utilize for this *HPX* *locality* (default: `all`, i.e., the number of cores is based on the number of threads `--hpx:threads` assuming `--hpx:bind=compact`.

`--hpx:affinity arg`
The affinity domain the OS threads will be confined to, possible values: `pu`, `core`, `numa`, `machine` (default: `pu`).

`--hpx:bind arg`
he detailed affinity description for the OS threads, see *More details about HPX command line options* for a detailed description of possible values. Do not use with `--hpx:pu-step`, `--hpx:pu-offset` or `--hpx:affinity` options. Implies `--hpx:numa-sensitive` (`--hpx:bind=none`) disables defining thread affinities).

`--hpx:use-process-mask`
Use the process mask to restrict available hardware resources (implies `--hpx:ignore-batch-env`).

`--hpx:print-bind`
Print to the console the bit masks calculated from the arguments specified to all `--hpx:bind` options.

--hpx:queuing arg

The queue scheduling policy to use. Options are `local`, `local-priority-fifo`, `local-priority-lifo`, `static`, `static-priority`, `abp-priority-fifo`, `local-workrequesting-fifo`, `local-workrequesting-lifo` `local-workrequesting-mc`, and `abp-priority-lifo` (default: `local-priority-fifo`).

--hpx:high-priority-threads arg

The number of operating system threads maintaining a high priority queue (default: number of OS threads), valid for *--hpx:queuing*=`abp-priority`, *--hpx:queuing*`static-priority` and *--hpx:queuing*`local-priority` only.

--hpx:numa-sensitive

Makes the scheduler NUMA sensitive.

HPX configuration options

--hpx:app-config arg

Load the specified application configuration (ini) file.

--hpx:config arg

Load the specified *HPX* configuration (ini) file.

--hpx:ini arg

Add a configuration definition to the default runtime configuration.

--hpx:exit

Exit after configuring the runtime.

HPX debugging options

--hpx:list-symbolic-names

List all registered symbolic names after startup.

--hpx:list-component-types

List all dynamic component types after startup.

--hpx:dump-config-initial

Print the initial runtime configuration.

--hpx:dump-config

Print the final runtime configuration.

--hpx:debug-hpx-log [arg]

Enable all messages on the *HPX* log channel and send all *HPX* logs to the target destination (default: `cout`).

--hpx:debug-agas-log [arg]

Enable all messages on the *AGAS* log channel and send all *AGAS* logs to the target destination (default: `cout`).

--hpx:debug-parcel-log [arg]

Enable all messages on the parcel transport log channel and send all parcel transport logs to the target destination (default: `cout`).

--hpx:debug-timing-log [arg]

Enable all messages on the timing log channel and send all timing logs to the target destination (default: `cout`).

`--hpx:debug-app-log [arg]`

Enable all messages on the application log channel and send all application logs to the target destination (default: `cout`).

`--hpx:debug-clp`

Debug command line processing.

`--hpx:attach-debugger arg`

Wait for a debugger to be attached, possible arg values: `startup` or `exception` (default: `startup`)

HPX options related to performance counters

`--hpx:print-counter`

Print the specified performance counter either repeatedly and/or at the times specified by *--hpx:print-counter-at* (see also option *--hpx:print-counter-interval*).

`--hpx:print-counter-reset`

Print the specified performance counter either repeatedly and/or at the times specified by *--hpx:print-counter-at*. Reset the counter after the value is queried (see also option *--hpx:print-counter-interval*).

`--hpx:print-counter-interval`

Print the performance counter(s) specified with *--hpx:print-counter* repeatedly after the time interval (specified in milliseconds), (default: `0`, which means print once at shutdown).

`--hpx:print-counter-destination`

Print the performance counter(s) specified with *--hpx:print-counter* to the given file (default: `console`).

`--hpx:list-counters`

List the names of all registered performance counters, possible values: `minimal` (prints counter name skeletons), `full` (prints all available counter names).

`--hpx:list-counter-infos`

List the description of all registered performance counters, possible values: `minimal` (prints info for counter name skeletons), `full` (prints all available counter infos).

`--hpx:print-counter-format`

Print the performance counter(s) specified with *--hpx:print-counter*. Possible formats in CSV include a format with a header or without any header (see option *--hpx:no-csv-header*). Possible values: `csv` (prints counter values in CSV format with full names as header), `csv-short` (prints counter values in CSV format with short names provided with *--hpx:print-counter* as *--hpx:print-counter* `shortname, full-countername`

`--hpx:no-csv-header`

Print the performance counter(s) specified with *--hpx:print-counter* and `csv` or `csv-short` format specified with *--hpx:print-counter-format* without header.

`--hpx:print-counter-at arg`

Print the performance counter(s) specified with *--hpx:print-counter* (or *--hpx:print-counter-reset*) at the given point in time, possible argument values: `startup`, `shutdown` (default), `noshutdown`.

`--hpx:reset-counters`

Reset all performance counter(s) specified with *--hpx:print-counter* after they have been evaluated.

--hpx:print-counters-locally

Each *locality* prints only its own local counters. If this is used with *--hpx:print-counter-destination*=<file>, the code will append a ".<locality_id>" to the file name in order to avoid clashes between localities.

Command line argument shortcuts

Additionally, the following shortcuts are available from every *HPX* application.

Table 2.9: Predefined command line option shortcuts

Shortcut option	Equivalent long option
-a	*--hpx:agas*
-c	*--hpx:console*
-h	*--hpx:help*
-I	*--hpx:ini*
-l	*--hpx:localities*
-p	*--hpx:app-config*
-q	*--hpx:queuing*
-r	*--hpx:run-agas-server*
-t	*--hpx:threads*
-v	*--hpx:version*
-w	*--hpx:worker*
-x	*--hpx:hpx*
-0	*--hpx:node*=0
-1	*--hpx:node*=1
-2	*--hpx:node*=2
-3	*--hpx:node*=3
-4	*--hpx:node*=4
-5	*--hpx:node*=5
-6	*--hpx:node*=6
-7	*--hpx:node*=7
-8	*--hpx:node*=8
-9	*--hpx:node*=9

Note: The short options listed above are disabled by default if the application is built using `#include <hpx/hpx_main.hpp>`. See *Re-use the main() function as the main HPX entry point* for more information. The rationale behind this is that in this case the user's application may handle its own command line options, since *HPX* passes all unknown options to `main()`. Short options like `-t` are prone to create ambiguities regarding what the application will support. Hence, the user should instead rely on the corresponding long options like `--hpx:threads` in such a case.

It is possible to define your own shortcut options. In fact, all of the shortcuts listed above are pre-defined using the technique described here. Also, it is possible to redefine any of the pre-defined shortcuts to expand differently as well.

Shortcut options are obtained from the internal configuration database. They are stored as key-value properties in a special properties section named `hpx.commandline`. You can define your own shortcuts by adding the corresponding definitions to one of the ini configuration files as described in the section *Configuring HPX applications*. For instance, in order to define a command line shortcut `--p`, which should expand to `-hpx:print-counter`, the following configuration information needs to be added to one of the ini configuration files:

```
[hpx.commandline.aliases]
--pc = --hpx:print-counter
```

Note: Any arguments for shortcut options passed on the command line are retained and passed as arguments to the corresponding expanded option. For instance, given the definition above, the command line option:

```
--pc=/threads{locality#0/total}/count/cumulative
```

would be expanded to:

```
--hpx:print-counter=/threads{locality#0/total}/count/cumulative
```

Important: Any shortcut option should either start with a single `'-'` or with two `'--'` characters. Shortcuts starting with a single `'-'` are interpreted as short options (i.e., everything after the first character following the `'-'` is treated as the argument). Shortcuts starting with `'--'` are interpreted as long options. No other shortcut formats are supported.

Specifying options for single localities only

For runs involving more than one *locality*, it is sometimes desirable to supply specific command line options to single localities only. When the *HPX* application is launched using a scheduler (like PBS; for more details see section *How to use HPX applications with PBS*), specifying dedicated command line options for single localities may be desirable. For this reason all of the command line options that have the general format `--hpx:<some_key>` can be used in a more general form: `--hpx:<N>:<some_key>`, where `<N>` is the number of the *locality* this command line option will be applied to; all other localities will simply ignore the option. For instance, the following PBS script passes the option `--hpx:pu-offset=4` to the *locality* `'1'` only.

```
#!/bin/bash
#
#PBS -l nodes=2:ppn=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

pbsdsh -u $APP_PATH $APP_OPTIONS --hpx:1:pu-offset=4 --hpx:nodes=`cat $PBS_NODEFILE`
```

Caution: If the first application specific argument (inside `$APP_OPTIONS`) is a non-option (i.e., does not start with a `-` or a `--`), then it must be placed before the option `--hpx:nodes`, which, in this case, should be the last option on the command line.

Alternatively, use the option `--hpx:endnodes` to explicitly mark the end of the list of node names:

```
$ pbsdsh -u $APP_PATH --hpx:1:pu-offset=4 --hpx:nodes=`cat $PBS_NODEFILE` --hpx:endnodes $APP_OPTIONS
```

More details about *HPX* command line options

This section documents the following list of the command line options in more detail:

- *The command line option --hpx:bind*

The command line option `--hpx:bind`

This command line option allows one to specify the required affinity of the *HPX* worker threads to the underlying processing units. As a result the worker threads will run only on the processing units identified by the corresponding bind specification. The affinity settings are to be specified using `--hpx:bind=<BINDINGS>`, where `<BINDINGS>` have to be formatted as described below.

In addition to the syntax described below, one can use `--hpx:bind=none` to disable all binding of any threads to a particular core. This is mostly supported for debugging purposes.

The specified affinities refer to specific regions within a machine hardware topology. In order to understand the hardware topology of a particular machine, it may be useful to run the lstopo tool, which is part of Portable Hardware Locality (HWLOC), to see the reported topology tree. Seeing and understanding a topology tree will definitely help in understanding the concepts that are discussed below.

Affinities can be specified using hwloc tuples. Tuples of hwloc *objects* and associated *indexes* can be specified in the form `object:index`, `object:index-index` or `object:index,...,index`. Hwloc objects represent types of mapped items in a topology tree. Possible values for objects are `socket`, `numanode`, `core` and `pu` (processing unit). Indexes are non-negative integers that specify a unique physical object in a topology tree using its logical sequence number.

Chaining multiple tuples together in the more general form `object1:index1[.object2:index2[...]]` is permissible. While the first tuple's object may appear anywhere in the topology, the Nth tuple's object must have a shallower topology depth than the (N+1)th tuple's object. Put simply: as you move right in a tuple chain, objects must go deeper in the topology tree. Indexes specified in chained tuples are relative to the scope of the parent object. For example, `socket:0.core:1` refers to the second core in the first socket (all indices are zero based).

Multiple affinities can be specified using several `--hpx:bind` command line options or by appending several affinities separated by a `';'`. By default, if multiple affinities are specified, they are added.

`"all"` is a special affinity consisting in the entire current topology.

Note: All "names" in an affinity specification, such as `thread`, `socket`, `numanode`, `pu` or `all`, can be abbreviated. Thus, the affinity specification `threads:0-3=socket:0.core:1.pu:1` is fully equivalent to its shortened form `t:0-3=s:0.c:1.p:1`.

Here is a full grammar describing the possible format of mappings:

```
mappings      ::=  distribution | mapping (";" mapping)*
distribution  ::=  "compact" | "scatter" | "balanced" | "numa-balanced"
mapping       ::=  thread_spec "=" pu_specs
thread_spec   ::=  "thread:" range_specs
pu_specs      ::=  pu_spec ("." pu_spec)*
pu_spec       ::=  type ":" range_specs | "~" pu_spec
range_specs   ::=  range_spec ("," range_spec)*
range_spec    ::=  int | int "-" int | "all"
type          ::=  "socket" | "numanode" | "core" | "pu"
```

The following example assumes a system with at least 4 cores, where each core has more than 1 processing unit (hardware threads). Running `hello_world_distributed` with 4 OS threads (on 4 processing units), where each of those threads is bound to the first processing unit of each of the cores, can be achieved by invoking:

```
$ hello_world_distributed -t4 --hpx:bind=thread:0-3=core:0-3.pu:0
```

Here, `thread:0-3` specifies the OS threads used to define affinity bindings, and `core:0-3.pu:` defines that for each of the cores (`core:0-3`) only their first processing unit `pu:0` should be used.

Note: The command line option *--hpx:print-bind* can be used to print the bitmasks generated from the affinity mappings as specified with *--hpx:bind*. For instance, on a system with hyperthreading enabled (i.e. 2 processing units per core), the command line:

```
$ hello_world_distributed -t4 --hpx:bind=thread:0-3=core:0-3.pu:0 --hpx:print-bind
```

will cause this output to be printed:

```
0: PU L#0(P#0), Core L#0, Socket L#0, Node L#0(P#0)
1: PU L#2(P#2), Core L#1, Socket L#0, Node L#0(P#0)
2: PU L#4(P#4), Core L#2, Socket L#0, Node L#0(P#0)
3: PU L#6(P#6), Core L#3, Socket L#0, Node L#0(P#0)
```

where each bit in the bitmasks corresponds to a processing unit the listed worker thread will be bound to run on.

The difference between the four possible predefined distribution schemes (`compact`, `scatter`, `balanced` and `numa-balanced`) is best explained with an example. Imagine that we have a system with 4 cores and 4 hardware threads per core on 2 sockets. If we place 8 threads the assignments produced by the `compact`, `scatter`, `balanced` and `numa-balanced` types are shown in the figure below. Notice that `compact` does not fully utilize all the cores in the system. For this reason it is recommended that applications are run using the `scatter` or `balanced`/`numa-balanced` options in most cases.

In addition to the predefined distributions it is possible to restrict the resources used by *HPX* to the process CPU mask. The CPU mask is typically set by e.g. MPI[57] and batch environments. Using the command line option *--hpx:use-process-mask* makes *HPX* act as if only the processing units in the CPU mask are available for use by *HPX*. The number of threads is automatically determined from the CPU mask. The number of threads can still be changed manually using this option, but only to a number less than or equal to the number of processing units in the CPU mask. The option *--hpx:print-bind* is useful in conjunction with *--hpx:use-process-mask* to make sure threads are placed as expected.

2.3.10 Writing single-node applications

Being a C++ Standard Library for Concurrency and Parallelism, *HPX* implements all of the corresponding facilities as defined by the C++ Standard but also those which are proposed as part of the ongoing C++ standardization process. This section focuses on the features available in *HPX* for parallel and concurrent computation on a single node, although many of the features presented here are also implemented to work in the distributed case.

[57] https://en.wikipedia.org/wiki/Message_Passing_Interface

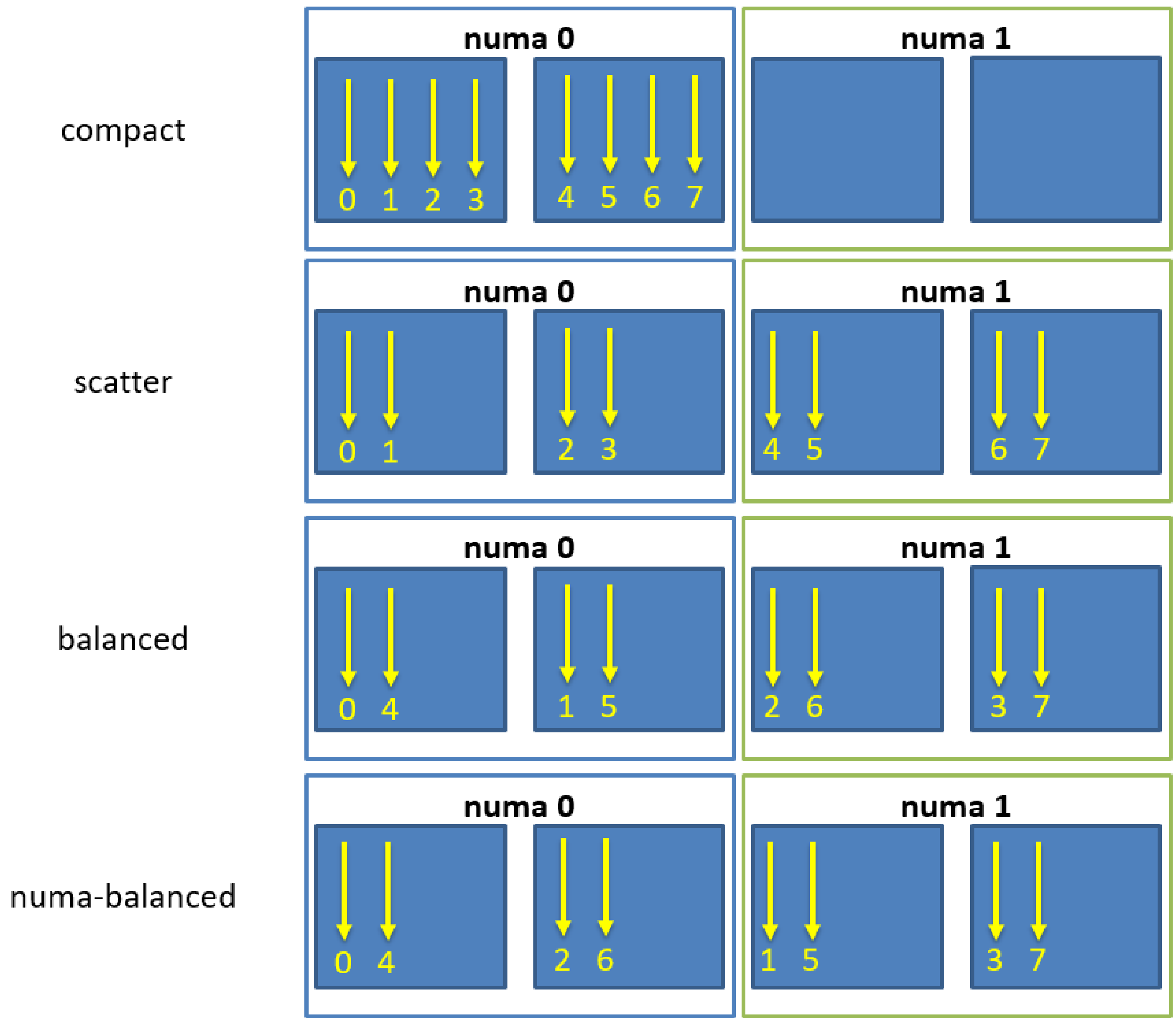

Fig. 2.7: Schematic of thread affinity type distributions.

Synchronization objects

The following objects are providing synchronization for *HPX* applications:

1. *Barrier*
2. *Condition variable*
3. *Latch*
4. *Mutex*
5. *Shared mutex*
6. *Semaphore*
7. *Composable guards*

Barrier

Barriers are used for synchronizing multiple threads. They provide a synchronization point, where all threads must wait until they have all reached the barrier, before they can continue execution. This allows multiple threads to work together to solve a common task, and ensures that no thread starts working on the next task until all threads have completed the current task. This ensures that all threads are in the same state before performing any further operations, leading to a more consistent and accurate computation.

Unlike latches, barriers are reusable: once the participating threads are released from a barrier's synchronization point, they can re-use the same barrier. It is thus useful for managing repeated tasks, or phases of a larger task, that are handled by multiple threads. The code below shows how barriers can be used to synchronize two threads:

```
#include <hpx/barrier.hpp>
#include <hpx/future.hpp>
#include <hpx/init.hpp>

#include <iostream>

int hpx_main()
{
    hpx::barrier b(2);

    hpx::future<void> f1 = hpx::async([&b]() {
        std::cout << "Thread 1 started." << std::endl;
        // Do some computation
        b.arrive_and_wait();
        // Continue with next task
        std::cout << "Thread 1 finished." << std::endl;
    });

    hpx::future<void> f2 = hpx::async([&b]() {
        std::cout << "Thread 2 started." << std::endl;
        // Do some computation
        b.arrive_and_wait();
        // Continue with next task
        std::cout << "Thread 2 finished." << std::endl;
    });
```

(continues on next page)

(continued from previous page)

```
    f1.get();
    f2.get();

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

In this example, two `hpx::future` objects are created, each representing a separate thread of execution. The `wait` function of the `hpx::barrier` object is called by each thread. The threads will wait at the barrier until both have reached it. Once both threads have reached the barrier, they can continue with their next task.

Condition variable

A *condition variable* is a synchronization primitive in *HPX* that allows a thread to wait for a specific condition to be satisfied before continuing execution. It is typically used in conjunction with a mutex or a lock to protect shared data that is being modified by multiple threads. Hence, it blocks one or more threads until another thread both modifies a shared variable (the condition) and notifies the `condition_variable`. The code below shows how two threads modifying the shared variable `data` can be synchronized using the `condition_variable`:

```
#include <hpx/condition_variable.hpp>
#include <hpx/init.hpp>
#include <hpx/mutex.hpp>
#include <hpx/thread.hpp>

#include <iostream>
#include <string>

hpx::condition_variable cv;
hpx::mutex m;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until the main thread signals that data is ready
    std::unique_lock<hpx::mutex> lk(m);
    cv.wait(lk, [] { return ready; });

    // Access the shared resource
    std::cout << "Worker thread: Processing data...\n";
    data = "Test data after";

    // Send data back to the main thread
    processed = true;
    std::cout << "Worker thread: data processing is complete\n";

```

(continues on next page)

(continued from previous page)

```
    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again
    lk.unlock();
    cv.notify_one();
}

int hpx_main()
{
    hpx::thread worker(worker_thread);

    // Do some work
    std::cout << "Main thread: Preparing data...\n";
    data = "Test data before";
    hpx::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Main thread: Data before processing = " << data << '\n';

    // Signal that data is ready and send data to worker thread
    {
        std::lock_guard<hpx::mutex> lk(m);
        ready = true;
        std::cout << "Main thread: Data is ready...\n";
    }
    cv.notify_one();

    // Wait for the worker thread to finish
    {
        std::unique_lock<hpx::mutex> lk(m);
        cv.wait(lk, [] { return processed; });
    }
    std::cout << "Main thread: Data after processing = " << data << '\n';
    worker.join();

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

The main thread of the code above starts by creating a worker thread and preparing the shared variable `data`. Once the data is ready, the main thread acquires a lock on the mutex `m` using `std::lock_guard<hpx::mutex> lk(m)` and sets the ready flag to true, then signals the worker thread to start processing by calling `cv.notify_one()`. The `cv.wait()` call in the main thread then blocks until the worker thread signals that processing is complete by setting the `processed` flag.

The worker thread starts by acquiring a lock on the mutex `m` to ensure exclusive access to the shared data. The `cv.wait()` call blocks the thread until the `ready` flag is set by the main thread. Once this is true, the worker thread accesses the shared data resource, processes it, and sets the `processed` flag to indicate completion. The mutex is then unlocked using `lk.unlock()` and the `cv.notify_one()` call signals the main thread to resume execution. Finally, the new `data` is printed by the main thread to the console.

Latch

A *latch* is a downward counter which can be used to synchronize threads. The value of the counter is initialized on creation. Threads may block on the latch until the counter is decremented to zero. There is no possibility to increase or reset the counter, which makes the latch a single-use barrier.

In *HPX*, a latch is implemented as a counting semaphore, which can be initialized with a specific count value and decremented each time a thread reaches the latch. When the count value reaches zero, all waiting threads are unblocked and allowed to continue execution. The code below shows how latch can be used to synchronize 16 threads:

```
std::ptrdiff_t num_threads = 16;

///////////////////////////////////////////////////////////////////////////////
void wait_for_latch(hpx::latch& l)
{
    l.arrive_and_wait();
}

///////////////////////////////////////////////////////////////////////////////
int hpx_main(hpx::program_options::variables_map& vm)
{
    num_threads = vm["num-threads"].as<std::ptrdiff_t>();

    hpx::latch l(num_threads + 1);

    std::vector<hpx::future<void>> results;
    for (std::ptrdiff_t i = 0; i != num_threads; ++i)
        results.push_back(hpx::async(&wait_for_latch, std::ref(l)));

    // Wait for all threads to reach this point.
    l.arrive_and_wait();

    hpx::wait_all(results);

    return hpx::local::finalize();
}
```

In the above code, the `hpx_main` function creates a latch object `l` with a count of `num_threads + 1` and `num_threads` number of threads using `hpx::async`. These threads call the `wait_for_latch` function and pass the reference to the latch object. In the `wait_for_latch` function, the thread calls the `arrive_and_wait` method on the latch, which decrements the count of the latch and causes the thread to wait until the count reaches zero. Finally, the main thread waits for all the threads to arrive at the latch by calling the `arrive_and_wait` method and then waits for all the threads to finish by calling the `hpx::wait_all` method.

Mutex

A *mutex* (short for "mutual exclusion") is a synchronization primitive in *HPX* used to control access to a shared resource, ensuring that only one thread can access it at a time. A mutex is used to protect data structures from race conditions and other synchronization-related issues. When a thread acquires a mutex, other threads that try to access the same resource will be blocked until the mutex is released. The code below shows the basic use of mutexes:

```
#include <hpx/future.hpp>
#include <hpx/init.hpp>
#include <hpx/mutex.hpp>

#include <iostream>

int hpx_main()
{
    hpx::mutex m;

    hpx::future<void> f1 = hpx::async([&m]() {
        std::scoped_lock sl(m);
        std::cout << "Thread 1 acquired the mutex" << std::endl;
    });

    hpx::future<void> f2 = hpx::async([&m]() {
        std::scoped_lock sl(m);
        std::cout << "Thread 2 acquired the mutex" << std::endl;
    });

    hpx::wait_all(f1, f2);

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

In this example, two *HPX* threads created using `hpx::async` are acquiring a `hpx::mutex m`. `std::scoped_lock sl(m)` is used to take ownership of the given mutex `m`. When control leaves the scope in which the `scoped_lock` object was created, the `scoped_lock` is destructed and the mutex is released.

> **Attention:** A common way to acquire and release mutexes is by using the function `m.lock()` before accessing the shared resource, and `m.unlock()` called after the access is complete. However, these functions may lead to deadlocks in case of exception(s). That is, if an exception happens when the mutex is locked then the code that unlocks the mutex will never be executed, the lock will remain held by the thread that acquired it, and other threads will be unable to access the shared resource. This can cause a deadlock if the other threads are also waiting to acquire the same lock. For this reason, we suggest you use `std::scoped_lock`, which prevents this issue by releasing the lock when control leaves the scope in which the `scoped_lock` object was created.

Shared mutex

A *shared mutex* is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In contrast to other mutex types which facilitate exclusive access, a `shared_mutex` has two levels of access:

- *Exclusive access* prevents any other thread from acquiring the mutex, just as with the normal mutex. It does not matter if the other thread tries to acquire shared or exclusive access.
- *Shared access* allows multiple threads to acquire the mutex, but all of them only in shared mode. Exclusive access is not granted until all of the previous shared holders have returned the mutex (typically, as long as an exclusive request is waiting, new shared ones are queued to be granted after the exclusive access).

Shared mutexes are especially useful when shared data can be safely read by any number of threads simultaneously, but a thread may only write the same data when no other thread is reading or writing at the same time. A typical scenario is a database: The data can be read simultaneously by different threads with no problem. However, modification of the database is critical: if some threads read data while another one is writing, the threads reading may receive inconsistent data. Hence, while a thread is writing, reading should not be allowed. After writing is complete, reads can occur simultaneously again. The code below shows how `shared_mutex` can be used to synchronize reads and writes:

```
int const writers = 3;
int const readers = 3;
int const cycles = 10;

using std::chrono::milliseconds;

int hpx_main()
{
    std::vector<hpx::thread> threads;
    std::atomic<bool> ready(false);
    hpx::shared_mutex stm;

    for (int i = 0; i < writers; ++i)
    {
        threads.emplace_back([&ready, &stm, i] {
            std::mt19937 urng(static_cast<std::uint32_t>(std::time(nullptr)));
            std::uniform_int_distribution<int> dist(1, 1000);

            while (!ready)
            { /*** wait... ***/
            }

            for (int j = 0; j < cycles; ++j)
            {
                // scope of unique_lock
                {
                    std::unique_lock<hpx::shared_mutex> ul(stm);

                    std::cout << "^^^ Writer " << i << " starting..."
                              << std::endl;
                    hpx::this_thread::sleep_for(milliseconds(dist(urng)));
                    std::cout << "vvv Writer " << i << " finished."
                              << std::endl;
                }
```

(continues on next page)

(continued from previous page)

```
                hpx::this_thread::sleep_for(milliseconds(dist(urng)));
            }
        });
    }

    for (int i = 0; i < readers; ++i)
    {
        int k = writers + i;
        threads.emplace_back([&ready, &stm, k, i] {
            HPX_UNUSED(k);
            std::mt19937 urng(static_cast<std::uint32_t>(std::time(nullptr)));
            std::uniform_int_distribution<int> dist(1, 1000);

            while (!ready)
            { /*** wait... ***/
            }

            for (int j = 0; j < cycles; ++j)
            {
                // scope of shared_lock
                {
                    std::shared_lock<hpx::shared_mutex> sl(stm);

                    std::cout << "Reader " << i << " starting..." << std::endl;
                    hpx::this_thread::sleep_for(milliseconds(dist(urng)));
                    std::cout << "Reader " << i << " finished." << std::endl;
                }
                hpx::this_thread::sleep_for(milliseconds(dist(urng)));
            }
        });
    }

    ready = true;
    for (auto& t : threads)
        t.join();

    return hpx::local::finalize();
}
```

The above code creates `writers` and `readers` threads, each of which will perform `cycles` of operations. Both the writer and reader threads use the `hpx::shared_mutex` object `stm` to synchronize access to a shared resource.

- For the writer threads, a `unique_lock` on the shared mutex is acquired before each write operation and is released after control leaves the scope in which the `unique_lock` object was created.
- For the reader threads, a `shared_lock` on the shared mutex is acquired before each read operation and is released after control leaves the scope in which the `shared_lock` object was created.

Before each operation, both the reader and writer threads sleep for a random time period, which is generated using a random number generator. The random time period simulates the processing time of the operation.

Semaphore

Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. The two types of semaphores are:

- counting semaphore: it has a counter that is bigger than zero. The counter is initialized in the constructor. Acquiring the semaphore decreases the counter and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore. Unlike `hpx::mutex`, an `hpx::counting_semaphore` is not bound to a thread, which means that the acquire and release call of a semaphore can happen on different threads.
- binary semaphore: it is an alias for a `hpx::counting_semaphore<1>`. In this case, the least maximal value is 1. `hpx::binary_semaphore` can be used to implement locks.

```
#include <hpx/init.hpp>
#include <hpx/semaphore.hpp>
#include <hpx/thread.hpp>

#include <iostream>

// initialize the semaphore with a count of 3
hpx::counting_semaphore<> semaphore(3);

void worker()
{
    semaphore.acquire();    // decrement the semaphore's count
    std::cout << "Entering critical section" << std::endl;
    hpx::this_thread::sleep_for(std::chrono::seconds(1));
    semaphore.release();    // increment the semaphore's count
    std::cout << "Exiting critical section" << std::endl;
}

int hpx_main()
{
    hpx::thread t1(worker);
    hpx::thread t2(worker);
    hpx::thread t3(worker);
    hpx::thread t4(worker);
    hpx::thread t5(worker);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

In this example, the counting semaphore is initialized to the value of 3. This means that up to 3 threads can access the critical section (the section of code inside the worker() function) at the same time. When a thread enters the critical section, it acquires the semaphore, which decrements the count, while when it exits the critical section, it releases the semaphore, incrementing thus the count. The worker() function simulates a critical section by acquiring the semaphore, sleeping for 1 second and then releasing the semaphore.

In the main function, 5 worker threads are created and started, each trying to enter the critical section. If the count of the semaphore is already 0, a worker will wait until another worker releases the semaphore (increasing its value).

Composable guards

Composable guards operate in a manner similar to locks, but are applied only to asynchronous functions. The guard (or guards) is automatically locked at the beginning of a specified task and automatically unlocked at the end. Because guards are never added to an existing task's execution context, the calling of guards is freely composable and can never deadlock.

To call an application with a single guard, simply declare the guard and call run_guarded() with a function (task):

```
hpx::lcos::local::guard gu;
run_guarded(gu,task);
```

If a single method needs to run with multiple guards, use a guard set:

```
std::shared_ptr<hpx::lcos::local::guard> gu1(new hpx::lcos::local::guard());
std::shared_ptr<hpx::lcos::local::guard> gu2(new hpx::lcos::local::guard());
gs.add(*gu1);
gs.add(*gu2);
run_guarded(gs,task);
```

Guards use two atomic operations (which are not called repeatedly) to manage what they do, so overhead should be extremely low.

Execution control

The following objects are providing control of the execution in *HPX* applications:

1. *Futures*
2. *Channels*
3. *Task blocks*
4. *Task groups*
5. *Threads*

Futures

Futures are a mechanism to represent the result of a potentially asynchronous operation. A future is a type that represents a value that will become available at some point in the future, and it can be used to write asynchronous and parallel code. Futures can be returned from functions that perform time-consuming operations, allowing the calling code to continue executing while the function performs its work. The value of the future is set when the operation completes and can be accessed later. Futures are used in *HPX* to write asynchronous and parallel code. Below is an example demonstrating different features of futures:

```
#include <hpx/assert.hpp>
#include <hpx/future.hpp>
#include <hpx/hpx_main.hpp>
#include <hpx/tuple.hpp>

#include <iostream>
#include <utility>

int main()
{
    // Asynchronous execution with futures
    hpx::future<void> f1 = hpx::async(hpx::launch::async, []() {});
    hpx::shared_future<int> f2 =
        hpx::async(hpx::launch::async, []() { return 42; });
    hpx::future<int> f3 =
        f2.then([](hpx::shared_future<int>&& f) { return f.get() * 3; });

    hpx::promise<double> p;
    auto f4 = p.get_future();
    HPX_ASSERT(!f4.is_ready());
    p.set_value(123.45);
    HPX_ASSERT(f4.is_ready());

    hpx::packaged_task<int()> t([]() { return 43; });
    hpx::future<int> f5 = t.get_future();
    HPX_ASSERT(!f5.is_ready());
    t();
    HPX_ASSERT(f5.is_ready());

    // Fire-and-forget
    hpx::post([]() {
        std::cout << "This will be printed later\n" << std::flush;
    });

    // Synchronous execution
    hpx::sync([]() {
        std::cout << "This will be printed immediately\n" << std::flush;
    });

    // Combinators
    hpx::future<double> f6 = hpx::async([]() { return 3.14; });
    hpx::future<double> f7 = hpx::async([]() { return 42.0; });
    std::cout
        << hpx::when_all(f6, f7)
               .then([](hpx::future<
                            hpx::tuple<hpx::future<double>, hpx::future<double>>>
                                f) {
                   hpx::tuple<hpx::future<double>, hpx::future<double>> t =
                       f.get();
                   double pi = hpx::get<0>(t).get();
                   double r = hpx::get<1>(t).get();
                   return pi * r * r;
               })
```

(continues on next page)

(continued from previous page)

```
                .get()
            << std::endl;

    // Easier continuations with dataflow; it waits for all future or
    // shared_future arguments before executing the continuation, and also
    // accepts non-future arguments
    hpx::future<double> f8 = hpx::async([]() { return 3.14; });
    hpx::future<double> f9 = hpx::make_ready_future(42.0);
    hpx::shared_future<double> f10 = hpx::async([]() { return 123.45; });
    hpx::future<hpx::tuple<double, double>> f11 = hpx::dataflow(
        [](hpx::future<double> a, hpx::future<double> b,
            hpx::shared_future<double> c, double d) {
            return hpx::make_tuple<>(a.get() + b.get(), c.get() / d);
        },
        f8, f9, f10, -3.9);

    // split_future gives a tuple of futures from a future of tuple
    hpx::tuple<hpx::future<double>, hpx::future<double>> f12 =
        hpx::split_future(std::move(f11));
    std::cout << hpx::get<1>(f12).get() << std::endl;

    return 0;
}
```

The first section of the main function demonstrates how to use futures for asynchronous execution. The first two lines create two futures, one for void and another for an integer, using the `hpx::async()` function. These futures are executed *asynchronously* in separate threads using the `hpx::launch::async` launch policy. The third future is created by *chaining* the second future using the `then()` member function. This future multiplies the result of the second future by 3.

The next part of the code demonstrates how to use *promises* and *packaged* tasks, which are constructs used for communicating data between threads. The `promise` class is used to store a value that can be retrieved *later* using a future. The `packaged_task` class represents a task that can be executed *asynchronously*, and its result can be obtained using a future. The last three lines create a packaged task that returns an integer, obtain its future, execute the task, and check whether the future is ready or not.

The code then demonstrates how to use the `hpx::post()` and `hpx::sync()` functions for *fire-and-forget* and *synchronous* execution, respectively. The `hpx::post()` function executes a given function *asynchronously* and *returns immediately* without waiting for the result. The `hpx::sync()` function executes a given function *synchronously* and *waits* for the result before returning.

Next the code demonstrates the use of *combinators*, which are higher-order functions that combine two or more futures into a single future. The `hpx::when_all()` function is used to combine two futures, which return double values, into a tuple of futures. The `then()` member function is then used to compute the area of a circle using the values of the two futures. The `get()` member function is used to retrieve the result of the computation.

The last section demonstrates the use of `hpx::dataflow()`, which is a higher-order function that waits for all the future or shared_future arguments to be ready before executing the continuation. The `hpx::make_ready_future()` function is used to create a future with a given value. The `hpx::split_future()` function is used to split a future of a tuple into a tuple of futures. The last line retrieves the value of the second future in the tuple using `hpx::get()` and prints it to the console.

Extended facilities for futures

Concurrency is about both decomposing and composing the program from the parts that work well individually and together. It is in the composition of connected and multicore components where today's C++ libraries are still lacking.

The functionality of std::future[59] offers a partial solution. It allows for the separation of the initiation of an operation and the act of waiting for its result; however, the act of waiting is synchronous. In communication-intensive code this act of waiting can be unpredictable, inefficient and simply frustrating. The example below illustrates a possible synchronous wait using futures:

```
#include <future>
using namespace std;
int main()
{
    future<int> f = async([]() { return 123; });
    int result = f.get(); // might block
}
```

For this reason, *HPX* implements a set of extensions to std::future[60] (as proposed by N4313[61]). This proposal introduces the following key asynchronous operations to *hpx::future*, *hpx::shared_future* and *hpx::async*, which enhance and enrich these facilities.

Table 2.11: Facilities extending `std::future`

Facility	Description
hpx::future::then	In asynchronous programming, it is very common for one asynchronous operation, on completion, to invoke a second operation and pass data to it. The current C++ standard does not allow one to register a continuation to a future. With `then`, instead of waiting for the result, a continuation is "attached" to the asynchronous operation, which is invoked when the result is ready. Continuations registered using then function will help to avoid blocking waits or wasting threads on polling, greatly improving the responsiveness and scalability of an application.
unwrapping constructor for *hpx::future*	In some scenarios, you might want to create a future that returns another future, resulting in nested futures. Although it is possible to write code to unwrap the outer future and retrieve the nested future and its result, such code is not easy to write because users must handle exceptions and it may cause a blocking call. Unwrapping can allow users to mitigate this problem by doing an asynchronous call to unwrap the outermost future.
`hpx::future::is_ready`	There are often situations where a `get()` call on a future may not be a blocking call, or is only a blocking call under certain circumstances. This function gives the ability to test for early completion and allows us to avoid associating a continuation, which needs to be scheduled with some non-trivial overhead and near-certain loss of cache efficiency.
hpx::make_ready_future	Some functions may know the value at the point of construction. In these cases the value is immediately available, but needs to be returned as a future. By using `hpx::make_ready_future` a future can be created that holds a pre-computed result in its shared state. In the current standard it is non-trivial to create a future directly from a value. First a promise must be created, then the promise is set, and lastly the future is retrieved from the promise. This can now be done with one operation.

The standard also omits the ability to compose multiple futures. This is a common pattern that is ubiquitous in other asynchronous frameworks and is absolutely necessary in order to make C++ a powerful asynchronous programming language. Not including these functions is synonymous to Boolean algebra without AND/OR.

[59] http://en.cppreference.com/w/cpp/thread/future
[60] http://en.cppreference.com/w/cpp/thread/future
[61] http://wg21.link/n4313

In addition to the extensions proposed by N4313[62], *HPX* adds functions allowing users to compose several futures in a more flexible way.

Table 2.12: Facilities for composing `hpx::futures`

Facility	Description
hpx::when_any, *hpx::when_any_n*	Asynchronously wait for at least one of multiple future or shared_future objects to finish.
hpx::wait_any, *hpx::wait_any_n*	Synchronously wait for at least one of multiple future or shared_future objects to finish.
hpx::when_all, *hpx::when_all_n*	Asynchronously wait for all future and shared_future objects to finish.
hpx::wait_all, *hpx::wait_all_n*	Synchronously wait for all future and shared_future objects to finish.
hpx::when_some, *hpx::when_some_n*	Asynchronously wait for multiple future and shared_future objects to finish.
hpx::wait_some, *hpx::wait_some_n*	Synchronously wait for multiple future and shared_future objects to finish.
hpx::when_each	Asynchronously wait for multiple future and shared_future objects to finish and call a function for each of the future objects as soon as it becomes ready.
hpx::wait_each, *hpx::wait_each_n*	Synchronously wait for multiple future and shared_future objects to finish and call a function for each of the future objects as soon as it becomes ready.

Channels

Channels combine communication (the exchange of a value) with synchronization (guaranteeing that two calculations (tasks) are in a known state). A channel can transport any number of values of a given type from a sender to a receiver:

```
    hpx::lcos::local::channel<int> c;
    hpx::future<int> f = c.get();
    HPX_ASSERT(!f.is_ready());
    c.set(42);
    HPX_ASSERT(f.is_ready());
    std::cout << f.get() << std::endl;
```

Channels can be handed to another thread (or in case of channel components, to other localities), thus establishing a communication channel between two independent places in the program:

```
void do_something(hpx::lcos::local::receive_channel<int> c,
    hpx::lcos::local::send_channel<> done)
{
    // prints 43
    std::cout << c.get(hpx::launch::sync) << std::endl;
    // signal back
    done.set();
}

void send_receive_channel()
{
    hpx::lcos::local::channel<int> c;
    hpx::lcos::local::channel<> done;
```

(continues on next page)

62 http://wg21.link/n4313

(continued from previous page)

```
    hpx::post(&do_something, c, done);

    // send some value
    c.set(43);
    // wait for thread to be done
    done.get().wait();
}
```

Note how `hpx::lcos::local::channel::get` without any arguments returns a future which is ready when a value has been set on the channel. The launch policy `hpx::launch::sync` can be used to make `hpx::lcos::local::channel::get` block until a value is set and return the value directly.

A channel component is created on one *locality* and can be sent to another *locality* using an action. This example also demonstrates how a channel can be used as a range of values:

```
// channel components need to be registered for each used type (not needed
// for hpx::lcos::local::channel)
HPX_REGISTER_CHANNEL(double)

void channel_sender(hpx::lcos::channel<double> c)
{
    for (double d : c)
        hpx::cout << d << std::endl;
}
HPX_PLAIN_ACTION(channel_sender)

void channel()
{
    // create the channel on this locality
    hpx::lcos::channel<double> c(hpx::find_here());

    // pass the channel to a (possibly remote invoked) action
    hpx::post(channel_sender_action(), hpx::find_here(), c);

    // send some values to the receiver
    std::vector<double> v = {1.2, 3.4, 5.0};
    for (double d : v)
        c.set(d);

    // explicitly close the communication channel (implicit at destruction)
    c.close();
}
```

Task blocks

Task blocks in *HPX* provide a way to structure and organize the execution of tasks in a parallel program, making it easier to manage dependencies between tasks. A task block actually is a group of tasks that can be executed in parallel. Tasks in a task block can depend on other tasks in the same task block. The task block allows the runtime to optimize the execution of tasks, by scheduling them in an optimal order based on the dependencies between them.

The `define_task_block`, `run` and the `wait` functions implemented based on N4755[63] are based on the `task_block` concept that is a part of the common subset of the Microsoft Parallel Patterns Library (PPL)[64] and the Intel Threading Building Blocks (TBB)[65] libraries.

These implementations adopt a simpler syntax than exposed by those libraries— one that is influenced by language-based concepts, such as spawn and sync from Cilk++[66] and async and finish from X10[67]. They improve on existing practice in the following ways:

- The exception handling model is simplified and more consistent with normal C++ exceptions.
- Most violations of strict fork-join parallelism can be enforced at compile time (with compiler assistance, in some cases).
- The syntax allows scheduling approaches other than child stealing.

Consider an example of a parallel traversal of a tree, where a user-provided function compute is applied to each node of the tree, returning the sum of the results:

```
template <typename Func>
int traverse(node& n, Func && compute)
{
    int left = 0, right = 0;
    define_task_block(
        [&](task_block<>& tr) {
            if (n.left)
                tr.run([&] { left = traverse(*n.left, compute); });
            if (n.right)
                tr.run([&] { right = traverse(*n.right, compute); });
        });

    return compute(n) + left + right;
}
```

The example above demonstrates the use of two of the functions, *`hpx::experimental::define_task_block`* and the *`hpx::experimental::task_block::run`* member function of a *`hpx::experimental::task_block`*.

The `task_block` function delineates a region in a program code potentially containing invocations of threads spawned by the `run` member function of the `task_block` class. The `run` function spawns an *HPX* thread, a unit of work that is allowed to execute in parallel with respect to the caller. Any parallel tasks spawned by `run` within the task block are joined back to a single thread of execution at the end of the `define_task_block`. `run` takes a user-provided function object `f` and starts it asynchronously—i.e., it may return before the execution of `f` completes. The *HPX* scheduler may choose to run `f` immediately or delay running `f` until compute resources become available.

A `task_block` can be constructed only by `define_task_block` because it has no public constructors. Thus, `run` can be invoked directly or indirectly only from a user-provided function passed to `define_task_block`:

[63] http://wg21.link/n4755
[64] https://msdn.microsoft.com/en-us/library/dd492418.aspx
[65] https://www.threadingbuildingblocks.org/
[66] https://software.intel.com/en-us/articles/intel-cilk-plus/
[67] https://x10-lang.org/

```
void g();

void f(task_block<>& tr)
{
    tr.run(g);          // OK, invoked from within task_block in h
}

void h()
{
    define_task_block(f);
}

int main()
{
    task_block<> tr;    // Error: no public constructor
    tr.run(g);          // No way to call run outside of a define_task_block
    return 0;
}
```

Extensions for task blocks

Using execution policies with task blocks

HPX implements some extensions for `task_block` beyond the actual standards proposal N4755[68]. The main addition is that a `task_block` can be invoked with an execution policy as its first argument, very similar to the parallel algorithms.

An execution policy is an object that expresses the requirements on the ordering of functions invoked as a consequence of the invocation of a task block. Enabling passing an execution policy to `define_task_block` gives the user control over the amount of parallelism employed by the created `task_block`. In the following example the use of an explicit `par` execution policy makes the user's intent explicit:

```
template <typename Func>
int traverse(node *n, Func&& compute)
{
    int left = 0, right = 0;

    define_task_block(
        execution::par,                 // execution::parallel_policy
        [&](task_block<>& tb) {
            if (n->left)
                tb.run([&] { left = traverse(n->left, compute); });
            if (n->right)
                tb.run([&] { right = traverse(n->right, compute); });
        });

    return compute(n) + left + right;
}
```

This also causes the *`hpx::experimental::task_block`* object to be a template in our implementation. The template argument is the type of the execution policy used to create the task block. The template argument defaults to *`hpx::execution::parallel_policy`*.

[68] http://wg21.link/n4755

HPX still supports calling *hpx::experimental::define_task_block* without an explicit execution policy. In this case the task block will run using the *hpx::execution::parallel_policy*.

HPX also adds the ability to access the execution policy that was used to create a given task_block.

Using executors to run tasks

Often, users want to be able to not only define an execution policy to use by default for all spawned tasks inside the task block, but also to customize the execution context for one of the tasks executed by task_block::run. Adding an optionally passed executor instance to that function enables this use case:

```
template <typename Func>
int traverse(node *n, Func&& compute)
{
    int left = 0, right = 0;

    define_task_block(
        execution::par,                 // execution::parallel_policy
        [&](auto& tb) {
            if (n->left)
            {
                // use explicitly specified executor to run this task
                tb.run(my_executor(), [&] { left = traverse(n->left, compute); });
            }
            if (n->right)
            {
                // use the executor associated with the par execution policy
                tb.run([&] { right = traverse(n->right, compute); });
            }
        });

    return compute(n) + left + right;
}
```

HPX still supports calling *hpx::experimental::task_block::run* without an explicit executor object. In this case the task will be run using the executor associated with the execution policy that was used to call *hpx::experimental::define_task_block*.

Task groups

A *task group* in *HPX* is a synchronization primitive that allows you to execute a group of tasks concurrently and wait for their completion before continuing. The tasks in an hpx::experimental::task_group can be added dynamically. This is the *HPX* implementation of *tbb::task_group* of the Intel Threading Building Blocks (TBB)[69] library.

The example below shows that to use a task group, you simply create an hpx::task_group object and add tasks to it using the run() method. Once all the tasks have been added, you can call the wait() method to synchronize the tasks and wait for them to complete.

```
#include <hpx/experimental/task_group.hpp>
#include <hpx/init.hpp>
```

(continues on next page)

69 https://www.threadingbuildingblocks.org/

(continued from previous page)

```
#include <iostream>

void task1()
{
    std::cout << "Task 1 executed." << std::endl;
}

void task2()
{
    std::cout << "Task 2 executed." << std::endl;
}

int hpx_main()
{
    hpx::experimental::task_group tg;

    tg.run(task1);
    tg.run(task2);

    tg.wait();

    std::cout << "All tasks finished!" << std::endl;

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

Note: *task groups* and *task blocks* are both ways to group and synchronize parallel tasks, but *task groups* are used to group multiple tasks together as a single unit, while *task blocks* are used to execute a loop in parallel, with each iteration of the loop executing in a separate task. If the difference is not clear yet, continue reading.

A *task group* is a construct that allows multiple parallel tasks to be grouped together as a single unit. The task group provides a way to synchronize all the tasks in the group before continuing with the rest of the program.

A *task block*, on the other hand, is a parallel loop construct that allows you to execute a loop in parallel, with each iteration of the loop executing in a separate task. The loop iterations are executed in a block, meaning that the loop body is executed as a single task.

Threads

A thread in *HPX* refers to a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space. These threads can communicate with each other through various means, such as futures or shared data structures.

The example below demonstrates how to launch multiple threads and synchronize them using a `hpx::latch` object. It also shows how to query the state of threads and wait for futures to complete.

```
#include <hpx/future.hpp>
#include <hpx/init.hpp>
#include <hpx/thread.hpp>

#include <functional>
#include <iostream>
#include <vector>

int const num_threads = 10;

///////////////////////////////////////////////////////////////////////////////
void wait_for_latch(hpx::latch& l)
{
    l.arrive_and_wait();
}

int hpx_main()
{
    // Spawn a couple of threads
    hpx::latch l(num_threads + 1);

    std::vector<hpx::future<void>> results;
    results.reserve(num_threads);

    for (int i = 0; i != num_threads; ++i)
        results.push_back(hpx::async(&wait_for_latch, std::ref(l)));

    // Allow spawned threads to reach latch
    hpx::this_thread::yield();

    // Enumerate all suspended threads
    hpx::threads::enumerate_threads(
        [](hpx::threads::thread_id_type id) -> bool {
            std::cout << "thread " << hpx::thread::id(id) << " is "
                      << hpx::threads::get_thread_state_name(
                             hpx::threads::get_thread_state(id))
                      << std::endl;
            return true;    // always continue enumeration
        },
        hpx::threads::thread_schedule_state::suspended);

    // Wait for all threads to reach this point.
    l.arrive_and_wait();
```

(continues on next page)

(continued from previous page)

```
    hpx::wait_all(results);

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

In more detail, the `wait_for_latch()` function is a simple helper function that waits for a `hpx::latch` object to be released. At this point we remind that `hpx::latch` is a synchronization primitive that allows multiple threads to wait for a common event to occur.

In the `hpx_main()` function, an `hpx::latch` object is created with a count of `num_threads + 1`, indicating that `num_threads` threads need to arrive at the latch before the latch is released. The loop that follows launches `num_threads` asynchronous operations, each of which calls the `wait_for_latch` function. The resulting futures are added to the vector.

After the threads have been launched, `hpx::this_thread::yield()` is called to give them a chance to reach the latch before the program proceeds. Then, the `hpx::threads::enumerate_threads` function prints the state of each suspended thread, while the next call of `l.arrive_and_wait()` waits for all the threads to reach the latch. Finally, `hpx::wait_all` is called to wait for all the futures to complete.

Hint: An advantage of using `hpx::thread` over other threading libraries is that it is optimized for high-performance parallelism, with support for lightweight threads and task scheduling to minimize thread overhead and maximize parallelism. Additionally, `hpx::thread` integrates seamlessly with other features of *HPX* such as futures, promises, and task groups, making it a powerful tool for parallel programming.

Checkout the examples of *Shared mutex*, *Condition variable*, *Semaphore* to see how *HPX* threads are used in combination with other features.

High level parallel facilities

In preparation for the upcoming C++ Standards, there are currently several proposals targeting different facilities supporting parallel programming. *HPX* implements (and extends) some of those proposals. This is well aligned with our strategy to align the APIs exposed from *HPX* with current and future C++ Standards.

At this point, *HPX* implements several of the C++ Standardization working papers, most notably N4409[70] (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4755[71] (Task Blocks), and N4406[72] (Parallel Algorithms Need Executors).

[70] http://wg21.link/n4409
[71] http://wg21.link/n4755
[72] http://wg21.link/n4406

Using parallel algorithms

A parallel algorithm is a function template declared in the namespace `hpx::parallel`.

All parallel algorithms are very similar in semantics to their sequential counterparts (as defined in the `namespace std`) with an additional formal template parameter named `ExecutionPolicy`. The execution policy is generally passed as the first argument to any of the parallel algorithms and describes the manner in which the execution of these algorithms may be parallelized and the manner in which they apply user-provided function objects.

The applications of function objects in parallel algorithms invoked with an execution policy object of type *hpx::execution::sequenced_policy* or *hpx::execution::sequenced_task_policy* execute in sequential order. For *hpx::execution::sequenced_policy* the execution happens in the calling thread.

The applications of function objects in parallel algorithms invoked with an execution policy object of type *hpx::execution::parallel_policy* or *hpx::execution::parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and are indeterminately sequenced within each thread.

Important: It is the caller's responsibility to ensure correctness, such as making sure that the invocation does not introduce data races or deadlocks.

The example below demonstrates how to perform a sequential and parallel *hpx::for_each* loop on a vector of integers.

```cpp
#include <hpx/algorithm.hpp>
#include <hpx/execution.hpp>
#include <hpx/init.hpp>

#include <iostream>
#include <vector>

int hpx_main()
{
    std::vector<int> v{1, 2, 3, 4, 5};

    auto print = [](const int& n) { std::cout << n << ' '; };

    std::cout << "Print sequential: ";
    hpx::for_each(v.begin(), v.end(), print);
    std::cout << '\n';

    std::cout << "Print parallel: ";
    hpx::for_each(hpx::execution::par, v.begin(), v.end(), print);
    std::cout << '\n';

    return hpx::local::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::local::init(hpx_main, argc, argv);
}
```

The above code uses `hpx::for_each` to print the elements of the vector `v{1, 2, 3, 4, 5}`. At first, `hpx::for_each()` is called without an execution policy, which means that it applies the lambda function `print` to each element in the vector sequentially. Hence, the elements are printed in order.

Next, `hpx::for_each()` is called with the `hpx::execution::par` execution policy, which applies the lambda function `print` to each element in the vector in parallel. Therefore, the output order of the elements in the vector is not deterministic and may vary from run to run.

Parallel exceptions

During the execution of a standard parallel algorithm, if temporary memory resources are required by any of the algorithms and no memory is available, the algorithm throws a `std::bad_alloc` exception.

During the execution of any of the parallel algorithms, if the application of a function object terminates with an uncaught exception, the behavior of the program is determined by the type of execution policy used to invoke the algorithm:

- If the execution policy object is of type *hpx::execution::parallel_unsequenced_policy*, *hpx::terminate* shall be called.
- If the execution policy object is of type *hpx::execution::sequenced_policy*, *hpx::execution::sequenced_task_policy*, *hpx::execution::parallel_policy*, or *hpx::execution::parallel_task_policy*, the execution of the algorithm terminates with an *hpx::exception_list* exception. All uncaught exceptions thrown during the application of user-provided function objects shall be contained in the *hpx::exception_list*.

For example, the number of invocations of the user-provided function object in for_each is unspecified. When *hpx::for_each* is executed sequentially, only one exception will be contained in the *hpx::exception_list* object.

These guarantees imply that, unless the algorithm has failed to allocate memory and terminated with `std::bad_alloc`, all exceptions thrown during the execution of the algorithm are communicated to the caller. It is unspecified whether an algorithm implementation will "forge ahead" after encountering and capturing a user exception.

The algorithm may terminate with the `std::bad_alloc` exception even if one or more user-provided function objects have terminated with an exception. For example, this can happen when an algorithm fails to allocate memory while creating or adding elements to the *hpx::exception_list* object.

Parallel algorithms

HPX provides implementations of the following parallel algorithms:

Table 2.13: Non-modifying parallel algorithms of header hpx/algorithm.hpp

Name	Description	C++ standard
hpx::adjacent_find	Computes the differences between adjacent elements in a range.	adjacent_find[73]
hpx::all_of	Checks if a predicate is `true` for all of the elements in a range.	all_any_none_of[74]
hpx::any_of	Checks if a predicate is `true` for any of the elements in a range.	all_any_none_of[75]
hpx::count	Returns the number of elements equal to a given value.	count[76]
hpx::count_if	Returns the number of elements satisfying a specific criteria.	count_if[77]
hpx::equal	Determines if two sets of elements are the same.	equal[78]
hpx::find	Finds the first element equal to a given value.	find[79]
hpx::find_end	Finds the last sequence of elements in a certain range.	find_end[80]
hpx::find_first_of	Searches for any one of a set of elements.	find_first_of[81]
hpx::find_if	Finds the first element satisfying a specific criteria.	find_if[82]
hpx::find_if_not	Finds the first element not satisfying a specific criteria.	find_if_not[83]
hpx::for_each	Applies a function to a range of elements.	for_each[84]
hpx::for_each_n	Applies a function to a number of elements.	for_each_n[85]
hpx::lexicographical_compare	Checks if a range of values is lexicographically less than another range of values.	lexicographical_compare[86]
hpx::mismatch	Finds the first position where two ranges differ.	mismatch[87]
hpx::none_of	Checks if a predicate is `true` for none of the elements in a range.	all_any_none_of[88]
hpx::search	Searches for a range of elements.	search[89]
hpx::search_n	Searches for a number consecutive copies of an element in a range.	search_n[90]

[73] http://en.cppreference.com/w/cpp/algorithm/adjacent_find
[74] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[75] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[76] http://en.cppreference.com/w/cpp/algorithm/count
[77] http://en.cppreference.com/w/cpp/algorithm/count_if
[78] http://en.cppreference.com/w/cpp/algorithm/equal
[79] http://en.cppreference.com/w/cpp/algorithm/find
[80] http://en.cppreference.com/w/cpp/algorithm/find_end
[81] http://en.cppreference.com/w/cpp/algorithm/find_first_of
[82] http://en.cppreference.com/w/cpp/algorithm/find_if
[83] http://en.cppreference.com/w/cpp/algorithm/find_if_not
[84] http://en.cppreference.com/w/cpp/algorithm/for_each
[85] http://en.cppreference.com/w/cpp/algorithm/for_each_n
[86] http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
[87] http://en.cppreference.com/w/cpp/algorithm/mismatch
[88] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[89] http://en.cppreference.com/w/cpp/algorithm/search
[90] http://en.cppreference.com/w/cpp/algorithm/search_n

Table 2.14: Modifying parallel algorithms of header hpx/algorithm.hpp

Name	Description	C++ standard
hpx::copy	Copies a range of elements to a new location.	exclusive_scan[91]
hpx::copy_n	Copies a number of elements to a new location.	copy_n[92]
hpx::copy_if	Copies the elements from a range to a new location for which the given predicate is `true`	copy[93]
hpx::move	Moves a range of elements to a new location.	move[94]
hpx::fill	Assigns a range of elements a certain value.	fill[95]
hpx::fill_n	Assigns a value to a number of elements.	fill_n[96]
hpx::generate	Saves the result of a function in a range.	generate[97]
hpx::generate_n	Saves the result of N applications of a function.	generate_n[98]
hpx::experimental::reduce_by_key	Performs an inclusive scan on consecutive elements with matching keys, with a reduction to output only the final sum for each key. The key sequence `{1,1,1,2,3,3,3,3,1}` and value sequence `{2,3,4,5,6,7,8,9,10}` would be reduced to `keys={1,2,3,1}`, `values={9,5,30,10}`.	
hpx::remove	Removes the elements from a range that are equal to the given value.	remove[99]
hpx::remove_if	Removes the elements from a range that are equal to the given predicate is `false`	remove[100]
hpx::remove_copy	Copies the elements from a range to a new location that are not equal to the given value.	remove_copy[101]
hpx::remove_copy_if	Copies the elements from a range to a new location for which the given predicate is `false`	remove_copy[102]
hpx::replace	Replaces all values satisfying specific criteria with another value.	replace[103]
hpx::replace_if	Replaces all values satisfying specific criteria with another value.	replace[104]
hpx::replace_copy	Copies a range, replacing elements satisfying specific criteria with another value.	replace_copy[105]
hpx::replace_copy_if	Copies a range, replacing elements satisfying specific criteria with another value.	replace_copy[106]
hpx::reverse	Reverses the order elements in a range.	reverse[107]
hpx::reverse_copy	Creates a copy of a range that is reversed.	reverse_copy[108]
hpx::rotate	Rotates the order of elements in a range.	rotate[109]
hpx::rotate_copy	Copies and rotates a range of elements.	rotate_copy[110]
hpx::shift_left	Shifts the elements in the range left by n positions.	shift_left[111]
hpx::shift_right	Shifts the elements in the range right by n positions.	shift_right[112]
hpx::swap_ranges	Swaps two ranges of elements.	swap_ranges[113]
hpx::transform	Applies a function to a range of elements.	transform[114]
hpx::unique	Eliminates all but the first element from every consecutive group of equivalent elements from a range.	unique[115]
hpx::unique_copy	Copies the elements from one range to another in such a way that there are no consecutive equal elements.	unique_copy[116]

Table 2.15: Set operations on sorted sequences of header hpx/algorithm.hpp

Name	Description	C++ standard
hpx::merge	Merges two sorted ranges.	merge[117]
hpx::inplace_merge	Merges two ordered ranges in-place.	inplace_merge[118]
hpx::includes	Returns true if one set is a subset of another.	includes[119]
hpx::set_difference	Computes the difference between two sets.	set_difference[120]
hpx::set_intersection	Computes the intersection of two sets.	set_intersection[121]
hpx::set_symmetric_difference	Computes the symmetric difference between two sets.	set_symmetric_difference[122]
hpx::set_union	Computes the union of two sets.	set_union[123]

Table 2.16: Heap operations of header hpx/algorithm.hpp

Name	Description	C++ standard
hpx::is_heap	Returns `true` if the range is max heap.	is_heap[124]
hpx::is_heap_until	Returns the first element that breaks a max heap.	is_heap_until[125]
hpx::make_heap	Constructs a max heap in the range [first, last).	make_heap[126]

[91] http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
[92] http://en.cppreference.com/w/cpp/algorithm/copy_n
[93] http://en.cppreference.com/w/cpp/algorithm/copy
[94] http://en.cppreference.com/w/cpp/algorithm/move
[95] http://en.cppreference.com/w/cpp/algorithm/fill
[96] http://en.cppreference.com/w/cpp/algorithm/fill_n
[97] http://en.cppreference.com/w/cpp/algorithm/generate
[98] http://en.cppreference.com/w/cpp/algorithm/generate_n
[99] http://en.cppreference.com/w/cpp/algorithm/remove
[100] http://en.cppreference.com/w/cpp/algorithm/remove
[101] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[102] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[103] http://en.cppreference.com/w/cpp/algorithm/replace
[104] http://en.cppreference.com/w/cpp/algorithm/replace
[105] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[106] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[107] http://en.cppreference.com/w/cpp/algorithm/reverse
[108] http://en.cppreference.com/w/cpp/algorithm/reverse_copy
[109] http://en.cppreference.com/w/cpp/algorithm/rotate
[110] http://en.cppreference.com/w/cpp/algorithm/rotate_copy
[111] http://en.cppreference.com/w/cpp/algorithm/shift_left
[112] http://en.cppreference.com/w/cpp/algorithm/shift_right
[113] http://en.cppreference.com/w/cpp/algorithm/swap_ranges
[114] http://en.cppreference.com/w/cpp/algorithm/transform
[115] http://en.cppreference.com/w/cpp/algorithm/unique
[116] http://en.cppreference.com/w/cpp/algorithm/unique_copy
[117] http://en.cppreference.com/w/cpp/algorithm/merge
[118] http://en.cppreference.com/w/cpp/algorithm/inplace_merge
[119] http://en.cppreference.com/w/cpp/algorithm/includes
[120] http://en.cppreference.com/w/cpp/algorithm/set_difference
[121] http://en.cppreference.com/w/cpp/algorithm/set_intersection
[122] http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
[123] http://en.cppreference.com/w/cpp/algorithm/set_union

Table 2.17: Minimum/maximum operations of header hpx/algorithm.hpp

Name	Description	C++ standard
hpx::max_element	Returns the largest element in a range.	max_element[127]
hpx::min_element	Returns the smallest element in a range.	min_element[128]
hpx::minmax_element	Returns the smallest and the largest element in a range.	minmax_element[129]

Table 2.18: Partitioning Operations of header hpx/algorithm.hpp

Name	Description	C++ standard
hpx::nth_element	Partially sorts the given range making sure that it is partitioned by the given element	nth_element[130]
hpx::is_partitioned	Returns `true` if each true element for a predicate precedes the false elements in a range.	is_partitioned[131]
hpx::partition	Divides elements into two groups without preserving their relative order.	partition[132]
hpx::partition_copy	Copies a range dividing the elements into two groups.	partition_copy[133]
hpx::stable_partition	Divides elements into two groups while preserving their relative order.	stable_partition[134]

Table 2.19: Sorting Operations of header hpx/algorithm.hpp

Name	Description	C++ standard
hpx::is_sorted	Returns `true` if each element in a range is sorted.	is_sorted[135]
hpx::is_sorted_until	Returns the first unsorted element.	is_sorted_until[136]
hpx::sort	Sorts the elements in a range.	sort[137]
hpx::stable_sort	Sorts the elements in a range, maintain sequence of equal elements.	stable_sort[138]
hpx::partial_sort	Sorts the first elements in a range.	partial_sort[139]
hpx::partial_sort_copy	Sorts the first elements in a range, storing the result in another range.	partial_sort_copy[140]
hpx::experimental::sort_by_key	Sorts one range of data using keys supplied in another range.	

[124] http://en.cppreference.com/w/cpp/algorithm/is_heap
[125] http://en.cppreference.com/w/cpp/algorithm/is_heap_until
[126] http://en.cppreference.com/w/cpp/algorithm/make_heap
[127] http://en.cppreference.com/w/cpp/algorithm/max_element
[128] http://en.cppreference.com/w/cpp/algorithm/min_element
[129] http://en.cppreference.com/w/cpp/algorithm/minmax_element
[130] http://en.cppreference.com/w/cpp/algorithm/nth_element
[131] http://en.cppreference.com/w/cpp/algorithm/is_partitioned
[132] http://en.cppreference.com/w/cpp/algorithm/partition
[133] http://en.cppreference.com/w/cpp/algorithm/partition_copy
[134] http://en.cppreference.com/w/cpp/algorithm/stable_partition

Table 2.20: Numeric Parallel Algorithms of header hpx/numeric.hpp

Name	Description	C++ standard
hpx::adjacent_difference	Calculates the difference between each element in an input range and the preceding element.	adjacent_difference[141]
hpx::exclusive_scan	Does an exclusive parallel scan over a range of elements.	exclusive_scan[142]
hpx::inclusive_scan	Does an inclusive parallel scan over a range of elements.	inclusive_scan[143]
hpx::reduce	Sums up a range of elements.	reduce[144]
hpx::transform_exclusive_scan	Does an exclusive parallel scan over a range of elements after applying a function.	transform_exclusive_scan[145]
hpx::transform_inclusive_scan	Does an inclusive parallel scan over a range of elements after applying a function.	transform_inclusive_scan[146]
hpx::transform_reduce	Sums up a range of elements after applying a function. Also, accumulates the inner products of two input ranges.	transform_reduce[147]

135 http://en.cppreference.com/w/cpp/algorithm/is_sorted
136 http://en.cppreference.com/w/cpp/algorithm/is_sorted_until
137 http://en.cppreference.com/w/cpp/algorithm/sort
138 http://en.cppreference.com/w/cpp/algorithm/stable_sort
139 http://en.cppreference.com/w/cpp/algorithm/partial_sort
140 http://en.cppreference.com/w/cpp/algorithm/partial_sort_copy
141 http://en.cppreference.com/w/cpp/algorithm/adjacent_difference
142 http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
143 http://en.cppreference.com/w/cpp/algorithm/inclusive_scan
144 http://en.cppreference.com/w/cpp/algorithm/reduce
145 http://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan
146 http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan
147 http://en.cppreference.com/w/cpp/algorithm/transform_reduce

Table 2.21: Dynamic Memory Management of header hpx/memory.hpp

Name	Description	C++ standard
hpx::destroy	Destroys a range of objects.	destroy[148]
hpx::destroy_n	Destroys a range of objects.	destroy_n[149]
hpx::uninitialized_copy	Copies a range of objects to an uninitialized area of memory.	uninitialized_copy[150]
hpx::uninitialized_copy_n	Copies a number of objects to an uninitialized area of memory.	uninitialized_copy_n[151]
hpx::uninitialized_default_construct	Copies a range of objects to an uninitialized area of memory.	uninitialized_default_construct[152]
hpx::uninitialized_default_construct_n	Copies a number of objects to an uninitialized area of memory.	uninitialized_default_construct_n[153]
hpx::uninitialized_fill	Copies an object to an uninitialized area of memory.	uninitialized_fill[154]
hpx::uninitialized_fill_n	Copies an object to an uninitialized area of memory.	uninitialized_fill_n[155]
hpx::uninitialized_move	Moves a range of objects to an uninitialized area of memory.	uninitialized_move[156]
hpx::uninitialized_move_n	Moves a number of objects to an uninitialized area of memory.	uninitialized_move_n[157]
hpx::uninitialized_value_construct	Constructs objects in an uninitialized area of memory.	uninitialized_value_construct[158]
hpx::uninitialized_value_construct_n	Constructs objects in an uninitialized area of memory.	uninitialized_value_construct_n[159]

Table 2.22: Index-based for-loops of header hpx/algorithm.hpp

Name	Description
hpx::experimental::for_loop	Implements loop functionality over a range specified by integral or iterator bounds.
hpx::experimental::for_loop_strided	Implements loop functionality over a range specified by integral or iterator bounds.
hpx::experimental::for_loop_n	Implements loop functionality over a range specified by integral or iterator bounds.
hpx::experimental::for_loop_n_strided	Implements loop functionality over a range specified by integral or iterator bounds.

[148] http://en.cppreference.com/w/cpp/memory/destroy
[149] http://en.cppreference.com/w/cpp/memory/destroy_n
[150] http://en.cppreference.com/w/cpp/memory/uninitialized_copy
[151] http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n
[152] http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct
[153] http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n
[154] http://en.cppreference.com/w/cpp/memory/uninitialized_fill
[155] http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n
[156] http://en.cppreference.com/w/cpp/memory/uninitialized_move
[157] http://en.cppreference.com/w/cpp/memory/uninitialized_move_n
[158] http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct
[159] http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct_n

Executor parameters and executor parameter traits

HPX introduces the notion of execution parameters and execution parameter traits. At this point, the only parameter that can be customized is the size of the chunks of work executed on a single *HPX* thread (such as the number of loop iterations combined to run as a single task).

An executor parameter object is responsible for exposing the calculation of the size of the chunks scheduled. It abstracts the (potentially platform-specific) algorithms of determining those chunk sizes.

The way executor parameters are implemented is aligned with the way executors are implemented. All functionalities of concrete executor parameter types are exposed and accessible through a corresponding customization point, e.g. `get_chunk_size()`.

With `executor_parameter_traits`, clients access all types of executor parameters uniformly, e.g.:

```
std::size_t chunk_size =
    hpx::execution::get_chunk_size(my_parameter, my_executor,
        num_cores, num_tasks);
```

This call synchronously retrieves the size of a single chunk of loop iterations (or similar) to combine for execution on a single *HPX* thread if the overall number of cores `num_cores` and tasks to schedule is given by `num_tasks`. The lambda function exposes a means of test-probing the execution of a single iteration for performance measurement purposes. The execution parameter type might dynamically determine the execution time of one or more tasks in order to calculate the chunk size; see `hpx::execution::experimental::auto_chunk_size` for an example of this executor parameter type.

Other functions in the interface exist to discover whether an executor parameter type should be invoked once (i.e., it returns a static chunk size; see `hpx::execution::experimental::static_chunk_size`) or whether it should be invoked for each scheduled chunk of work (i.e., it returns a variable chunk size; for an example, see `hpx::execution::experimental::guided_chunk_size`).

Although this interface appears to require executor parameter type authors to implement all different basic operations, none are required. In practice, all operations have sensible defaults. However, some executor parameter types will naturally specialize all operations for maximum efficiency.

HPX implements the following executor parameter types:

- `hpx::execution::experimental::auto_chunk_size`: Loop iterations are divided into pieces and then assigned to threads. The number of loop iterations combined is determined based on measurements of how long the execution of 1% of the overall number of iterations takes. This executor parameter type makes sure that as many loop iterations are combined as necessary to run for the amount of time specified.
- `hpx::execution::experimental::static_chunk_size`: Loop iterations are divided into pieces of a given size and then assigned to threads. If the size is not specified, the iterations are, if possible, evenly divided contiguously among the threads. This executor parameters type is equivalent to OpenMP's STATIC scheduling directive.
- `hpx::execution::experimental::dynamic_chunk_size`: Loop iterations are divided into pieces of a given size and then dynamically scheduled among the cores; when a core finishes one chunk, it is dynamically assigned another. If the size is not specified, the default chunk size is 1. This executor parameter type is equivalent to OpenMP's DYNAMIC scheduling directive.
- `hpx::execution::experimental::guided_chunk_size`: Iterations are dynamically assigned to cores in blocks as cores request them until no blocks remain to be assigned. This is similar to `dynamic_chunk_size` except that the block size decreases each time a number of loop iterations is given to a thread. The size of the initial block is proportional to `number_of_iterations / number_of_cores`. Subsequent blocks are proportional to `number_of_iterations_remaining / number_of_cores`. The optional chunk size parameter defines the minimum block size. The default minimal chunk size is 1. This executor parameter type is equivalent to OpenMP's GUIDED scheduling directive.

2.3.11 Writing distributed applications

This section focuses on the features of *HPX* needed to write distributed applications, namely the *Active Global Address Space* (*AGAS*), remotely executable functions (i.e., *actions*), and distributed objects (i.e., *components*).

Global names

HPX implements an *Active Global Address Space* (*AGAS*) which exposes a single uniform address space spanning all localities an application runs on. AGAS is a fundamental component of the ParalleX execution model. Conceptually, there is no rigid demarcation of local or global memory in AGAS; all available memory is a part of the same address space. AGAS enables named objects to be moved (migrated) across localities without having to change the object's name; i.e., no references to migrated objects have to be ever updated. This feature has significance for dynamic load balancing and in applications where the workflow is highly dynamic, allowing work to be migrated from heavily loaded nodes to less loaded nodes. In addition, immutability of names ensures that AGAS does not have to keep extra indirections ("bread crumbs") when objects move, hence, minimizing complexity of code management for system developers as well as minimizing overheads in maintaining and managing aliases.

The AGAS implementation in *HPX* does not automatically expose every local address to the global address space. It is the responsibility of the programmer to explicitly define which of the objects have to be globally visible and which of the objects are purely local.

In *HPX* global addresses (global names) are represented using the `hpx::id_type` data type. This data type is conceptually very similar to `void*` pointers as it does not expose any type information of the object it is referring to.

The only predefined global addresses are assigned to all localities. The following *HPX* API functions allow one to retrieve the global addresses of localities:

- `hpx::find_here`: retrieves the global address of the *locality* this function is called on.
- `hpx::find_all_localities`: retrieves the global addresses of all localities available to this application (including the *locality* the function is being called on).
- `hpx::find_remote_localities`: retrieves the global addresses of all remote localities available to this application (not including the *locality* the function is being called on).
- `hpx::get_num_localities`: retrieves the number of localities available to this application.
- `hpx::find_locality`: retrieves the global address of any *locality* supporting the given component type.
- `hpx::get_colocation_id`: retrieves the global address of the *locality* currently hosting the object with the given global address.

Additionally, the global addresses of localities can be used to create new instances of components using the following *HPX* API function:

- `hpx::components::new_`: Creates a new instance of the given `Component` type on the specified *locality*.

Note: *HPX* does not expose any functionality to delete component instances. All global addresses (as represented using `hpx::id_type`) are automatically garbage collected. When the last (global) reference to a particular component instance goes out of scope, the corresponding component instance is automatically deleted.

Posting actions

Action type definition

Actions are special types used to describe possibly remote operations. For every global function and every member function which has to be invoked distantly, a special type must be defined. For any global function the special macro *HPX_PLAIN_ACTION* can be used to define the action type. Here is an example demonstrating this:

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }
}

// This will define the action type 'some_global_action' which represents
// the function 'app::some_global_function'.
HPX_PLAIN_ACTION(app::some_global_function, some_global_action);
```

Important: The macro *HPX_PLAIN_ACTION* has to be placed in global namespace, even if the wrapped function is located in some other namespace. The newly defined action type is placed in the global namespace as well.

If the action type should be defined somewhere not in global namespace, the action type definition has to be split into two macro invocations (*HPX_DEFINE_PLAIN_ACTION* and *HPX_REGISTER_ACTION*) as shown in the next example:

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }

    // On conforming compilers the following macro expands to:
    //
    //    typedef hpx::actions::make_action<
    //        decltype(&some_global_function), &some_global_function
    //    >::type some_global_action;
    //
    // This will define the action type 'some_global_action' which represents
    // the function 'some_global_function'.
    HPX_DEFINE_PLAIN_ACTION(some_global_function, some_global_action);
}

// The following macro expands to a series of definitions of global objects
// which are needed for proper serialization and initialization support
// enabling the remote invocation of the function``some_global_function``
HPX_REGISTER_ACTION(app::some_global_action, app_some_global_action);
```

The shown code defines an action type `some_global_action` inside the namespace `app`.

Important: If the action type definition is split between two macros as shown above, the name of the action type to create has to be the same for both macro invocations (here `some_global_action`).

Important: The second argument passed to *HPX_REGISTER_ACTION* (`app_some_global_action`) has to comprise a globally unique C++ identifier representing the action. This is used for serialization purposes.

For member functions of objects which have been registered with AGAS (e.g., 'components'), a different registration macro *HPX_DEFINE_COMPONENT_ACTION* has to be utilized. Any component needs to be declared in a header file and have some special support macros defined in a source file. Here is an example demonstrating this. The first snippet has to go into the header file:

```
namespace app
{
    struct some_component
      : hpx::components::component_base<some_component>
    {
        int some_member_function(std::string s)
        {
            return boost::lexical_cast<int>(s);
        }

        // This will define the action type 'some_member_action' which
        // represents the member function 'some_member_function' of the
        // object type 'some_component'.
        HPX_DEFINE_COMPONENT_ACTION(some_component, some_member_function,
            some_member_action);
    };
}

// Note: The second argument to the macro below has to be systemwide-unique
//       C++ identifiers
HPX_REGISTER_ACTION_DECLARATION(app::some_component::some_member_action, some_component_
↪some_action);
```

The next snippet belongs in a source file (e.g., the main application source file) in the simplest case:

```
typedef hpx::components::component<app::some_component> component_type;
typedef app::some_component some_component;

HPX_REGISTER_COMPONENT(component_type, some_component);

// The parameters for this macro have to be the same as used in the corresponding
// HPX_REGISTER_ACTION_DECLARATION() macro invocation above
typedef some_component::some_member_action some_component_some_action;
HPX_REGISTER_ACTION(some_component_some_action);
```

While these macro invocations are a bit more complex than those for simple global functions, they should still be manageable.

The most important macro invocation is the *HPX_DEFINE_COMPONENT_ACTION* in the header file as this defines the action type we need to invoke the member function. For a complete example of a simple component action see `component_in_executable.cpp`.

Action invocation

The process of invoking a global function (or a member function of an object) with the help of the associated action is called 'posting the action'. Actions can have arguments, which will be supplied while the action is applied. At the minimum, one parameter is required to `post` any action - the id of the *locality* the associated function should be invoked on (for global functions), or the id of the component instance (for member functions). Generally, *HPX* provides several ways to `post` an action, all of which are described in the following sections.

Generally, *HPX* actions are very similar to 'normal' C++ functions except that actions can be invoked remotely. Fig. 2.8 below shows an overview of the main API exposed by HPX. This shows the function invocation syntax as defined by the C++ language (dark gray), the additional invocation syntax as provided through C++ Standard Library features (medium gray), and the extensions added by *HPX* (light gray) where:

- `f` function to invoke,
- `p..`: (optional) arguments,
- R: return type of `f`,
- `action`: action type defined by, *HPX_DEFINE_PLAIN_ACTION* or *HPX_DEFINE_COMPONENT_ACTION* encapsulating `f`,
- `a`: an instance of the type `action`,
- `id`: the global address the action is applied to.

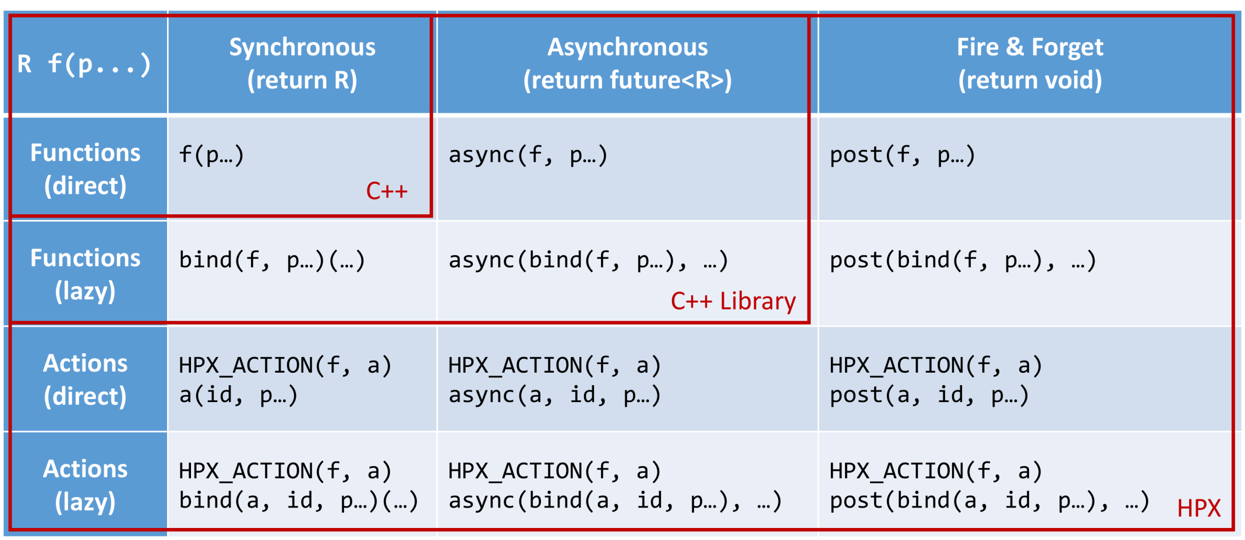

Fig. 2.8: Overview of the main API exposed by *HPX*.

This figure shows that *HPX* allows the user to `post` actions with a syntax similar to the C++ standard. In fact, all action types have an overloaded function operator allowing to synchronously `post` the action. Further, *HPX* implements `hpx::async` which semantically works similar to the way `std::async` works for plain C++ function.

Note: The similarity of posting an action to conventional function invocations extends even further. *HPX* implements `hpx::bind` and `hpx::function` two facilities which are semantically equivalent to the `std::bind` and `std::function` types as defined by the C++11 Standard. While `hpx::async` extends beyond the conventional semantics by supporting actions and conventional C++ functions, the *HPX* facilities `hpx::bind` and `hpx::function` extend beyond the conventional standard facilities too. The *HPX* facilities not only support conventional functions, but can be used for actions as well.

Additionally, *HPX* exposes `hpx::post` and `hpx::async_continue` both of which refine and extend the standard C++ facilities.

The different ways to invoke a function in *HPX* will be explained in more detail in the following sections.

Posting an action asynchronously without any synchronization

This method ('fire and forget') will make sure the function associated with the action is scheduled to run on the target *locality*. Posting the action does not wait for the function to start running, instead it is a fully asynchronous operation. The following example shows how to `post` the action as defined *in the previous section* on the local *locality* (the *locality* this code runs on):

```
some_global_action act;     // define an instance of some_global_action
hpx::post(act, hpx::find_here(), 2.0);
```

(the function `hpx::find_here()` returns the id of the local *locality*, i.e. the *locality* this code executes on).

Any component member function can be invoked using the same syntactic construct. Given that `id` is the global address for a component instance created earlier, this invocation looks like:

```
some_component_action act;     // define an instance of some_component_action
hpx::post(act, id, "42");
```

In this case any value returned from this action (e.g. in this case the integer `42` is ignored. Please look at *Action type definition* for the code defining the component action `some_component_action` used.

Posting an action asynchronously with synchronization

This method will make sure the action is scheduled to run on the target *locality*. Posting the action itself does not wait for the function to start running or to complete, instead this is a fully asynchronous operation similar to using `hpx::post` as described above. The difference is that this method will return an instance of a `hpx::future<>` encapsulating the result of the (possibly remote) execution. The future can be used to synchronize with the asynchronous operation. The following example shows how to `post` the action from above on the local *locality*:

```
some_global_action act;     // define an instance of some_global_action
hpx::future<void> f = hpx::async(act, hpx::find_here(), 2.0);
//
// ... other code can be executed here
//
f.get();    // this will possibly wait for the asynchronous operation to 'return'
```

(as before, the function `hpx::find_here()` returns the id of the local *locality* (the *locality* this code is executed on).

Note: The use of a `hpx::future<void>` allows the current thread to synchronize with any remote operation not returning any value.

Note: Any `std::future<>` returned from `std::async()` is required to block in its destructor if the value has not been set for this future yet. This is not true for `hpx::future<>` which will never block in its destructor, even if the value has not been returned to the future yet. We believe that consistency in the behavior of futures is more important than standards conformance in this case.

Any component member function can be invoked using the same syntactic construct. Given that `id` is the global address for a component instance created earlier, this invocation looks like:

```
some_component_action act;     // define an instance of some_component_action
hpx::future<int> f = hpx::async(act, id, "42");
//
// ... other code can be executed here
//
cout << f.get();    // this will possibly wait for the asynchronous operation to 'return'
↪42
```

Note: The invocation of `f.get()` will return the result immediately (without suspending the calling thread) if the result from the asynchronous operation has already been returned. Otherwise, the invocation of `f.get()` will suspend the execution of the calling thread until the asynchronous operation returns its result.

Posting an action synchronously

This method will schedule the function wrapped in the specified action on the target *locality*. While the invocation appears to be synchronous (as we will see), the calling thread will be suspended while waiting for the function to return. Invoking a plain action (e.g. a global function) synchronously is straightforward:

```
some_global_action act;     // define an instance of some_global_action
act(hpx::find_here(), 2.0);
```

While this call looks just like a normal synchronous function invocation, the function wrapped by the action will be scheduled to run on a new thread and the calling thread will be suspended. After the new thread has executed the wrapped global function, the waiting thread will resume and return from the synchronous call.

Equivalently, any action wrapping a component member function can be invoked synchronously as follows:

```
some_component_action act;     // define an instance of some_component_action
int result = act(id, "42");
```

The action invocation will either schedule a new thread locally to execute the wrapped member function (as before, `id` is the global address of the component instance the member function should be invoked on), or it will send a parcel to the remote *locality* of the component causing a new thread to be scheduled there. The calling thread will be suspended until the function returns its result. This result will be returned from the synchronous action invocation.

It is very important to understand that this 'synchronous' invocation syntax in fact conceals an asynchronous function call. This is beneficial as the calling thread is suspended while waiting for the outcome of a potentially remote operation. The *HPX* thread scheduler will schedule other work in the meantime, allowing the application to make further progress while the remote result is computed. This helps overlapping computation with communication and hiding communication latencies.

Note: The syntax of posting an action is always the same, regardless whether the target *locality* is remote to the invocation *locality* or not. This is a very important feature of *HPX* as it frees the user from the task of keeping track what actions have to be applied locally and which actions are remote. If the target for posting an action is local, a new thread is automatically created and scheduled. Once this thread is scheduled and run, it will execute the function encapsulated by that action. If the target is remote, *HPX* will send a parcel to the remote *locality* which encapsulates the action and its parameters. Once the parcel is received on the remote *locality* *HPX* will create and schedule a new thread there. Once this thread runs on the remote *locality*, it will execute the function encapsulated by the action.

Posting an action with a continuation but without any synchronization

This method is very similar to the method described in section *Posting an action asynchronously without any synchronization*. The difference is that it allows the user to chain a sequence of asynchronous operations, while handing the (intermediate) results from one step to the next step in the chain. Where `hpx::post` invokes a single function using 'fire and forget' semantics, `hpx::post_continue` asynchronously triggers a chain of functions without the need for the execution flow 'to come back' to the invocation site. Each of the asynchronous functions can be executed on a different *locality*.

Posting an action with a continuation and with synchronization

This method is very similar to the method described in section *Posting an action asynchronously with synchronization*. In addition to what `hpx::async` can do, the functions `hpx::async_continue` takes an additional function argument. This function will be called as the continuation of the executed action. It is expected to perform additional operations and to make sure that a result is returned to the original invocation site. This method chains operations asynchronously by providing a continuation operation which is automatically executed once the first action has finished executing.

As an example we chain two actions, where the result of the first action is forwarded to the second action and the result of the second action is sent back to the original invocation site:

```
// first action
std::int32_t action1(std::int32_t i)
{
    return i+1;
}
HPX_PLAIN_ACTION(action1);    // defines action1_type

// second action
std::int32_t action2(std::int32_t i)
{
    return i*2;
}
HPX_PLAIN_ACTION(action2);    // defines action2_type

// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;     // define an instance of 'action1_type'
action2_type act2;     // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1, hpx::make_continuation(act2),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n";   // will print: 86 ((42 + 1) * 2)
```

By default, the continuation is executed on the same *locality* as `hpx::async_continue` is invoked from. If you want to specify the *locality* where the continuation should be executed, the code above has to be written as:

```
// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;     // define an instance of 'action1_type'
action2_type act2;     // define an instance of 'action2_type'
hpx::future<int> f =
```

(continues on next page)

(continued from previous page)

```
    hpx::async_continue(act1, hpx::make_continuation(act2, hpx::find_here()),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n";   // will print: 86 ((42 + 1) * 2)
```

Similarly, it is possible to chain more than 2 operations:

```
action1_type act1;     // define an instance of 'action1_type'
action2_type act2;     // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1,
        hpx::make_continuation(act2, hpx::make_continuation(act1)),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n";   // will print: 87 ((42 + 1) * 2 + 1)
```

The function `hpx::make_continuation` creates a special function object which exposes the following prototype:

```
struct continuation
{
    template <typename Result>
    void operator()(hpx::id_type id, Result&& result) const
    {
        ...
    }
};
```

where the parameters passed to the overloaded function operator `operator()()` are:

- the `id` is the global id where the final result of the asynchronous chain of operations should be sent to (in most cases this is the id of the `hpx::future` returned from the initial call to `hpx::async_continue`. Any custom continuation function should make sure this `id` is forwarded to the last operation in the chain.
- the `result` is the result value of the current operation in the asynchronous execution chain. This value needs to be forwarded to the next operation.

Note: All of those operations are implemented by the predefined continuation function object which is returned from `hpx::make_continuation`. Any (custom) function object used as a continuation should conform to the same interface.

Action error handling

Like in any other asynchronous invocation scheme it is important to be able to handle error conditions occurring while the asynchronous (and possibly remote) operation is executed. In *HPX* all error handling is based on standard C++ exception handling. Any exception thrown during the execution of an asynchronous operation will be transferred back to the original invocation *locality*, where it is rethrown during synchronization with the calling thread.

Important: Exceptions thrown during asynchronous execution can be transferred back to the invoking thread only for the synchronous and the asynchronous case with synchronization. Like with any other unhandled exception, any exception thrown during the execution of an asynchronous action *without* synchronization will result in calling `hpx::terminate` causing the running application to exit immediately.

Note: Even if error handling internally relies on exceptions, most of the API functions exposed by *HPX* can be used without throwing an exception. Please see *Working with exceptions* for more information.

As an example, we will assume that the following remote function will be executed:

```
namespace app
{
    void some_function_with_error(int arg)
    {
        if (arg < 0) {
            HPX_THROW_EXCEPTION(hpx::error::bad_parameter,
                "some_function_with_error",
                "some really bad error happened");
        }
        // do something else...
    }
}

// This will define the action type 'some_error_action' which represents
// the function 'app::some_function_with_error'.
HPX_PLAIN_ACTION(app::some_function_with_error, some_error_action);
```

The use of `HPX_THROW_EXCEPTION` to report the error encapsulates the creation of a `hpx::exception` which is initialized with the error code `hpx::error::bad_parameter`. Additionally it carries the passed strings, the information about the file name, line number, and call stack of the point the exception was thrown from.

We invoke this action using the synchronous syntax as described before:

```
// note: wrapped function will throw hpx::exception
some_error_action act;            // define an instance of some_error_action
try {
    act(hpx::find_here(), -3);    // exception will be rethrown from here
}
catch (hpx::exception const& e) {
    // prints: 'some really bad error happened: HPX(bad parameter)'
    cout << e.what();
}
```

If this action is invoked asynchronously with synchronization, the exception is propagated to the waiting thread as well and is re-thrown from the future's function `get()`:

```
// note: wrapped function will throw hpx::exception
some_error_action act;            // define an instance of some_error_action
hpx::future<void> f = hpx::async(act, hpx::find_here(), -3);
try {
    f.get();                      // exception will be rethrown from here
}
catch (hpx::exception const& e) {
    // prints: 'some really bad error happened: HPX(bad parameter)'
    cout << e.what();
}
```

For more information about error handling please refer to the section *Working with exceptions*. There we also explain how to handle error conditions without having to rely on exception.

Writing components

A component in *HPX* is a C++ class which can be created remotely and for which its member functions can be invoked remotely as well. The following sections highlight how components can be defined, created, and used.

Defining components

In order for a C++ class type to be managed remotely in *HPX*, the type must be derived from the `hpx::components::component_base` template type. We call such C++ class types 'components'.

Note that the component type itself is passed as a template argument to the base class:

```
// header file some_component.hpp

#include <hpx/include/components.hpp>

namespace app
{
    // Define a new component type 'some_component'
    struct some_component
      : hpx::components::component_base<some_component>
    {
        // This member function is has to be invoked remotely
        int some_member_function(std::string const& s)
        {
            return boost::lexical_cast<int>(s);
        }

        // This will define the action type 'some_member_action' which
        // represents the member function 'some_member_function' of the
        // object type 'some_component'.
        HPX_DEFINE_COMPONENT_ACTION(some_component, some_member_function, some_member_
↪action);
    };
}

// This will generate the necessary boiler-plate code for the action allowing
// it to be invoked remotely. This declaration macro has to be placed in the
// header file defining the component itself.
//
// Note: The second argument to the macro below has to be systemwide-unique
//       C++ identifiers
//
HPX_REGISTER_ACTION_DECLARATION(app::some_component::some_member_action, some_component_
↪some_action);
```

There is more boiler plate code which has to be placed into a source file in order for the component to be usable. Every component type is required to have macros placed into its source file, one for each component type and one macro for each of the actions defined by the component type.

For instance:

```
// source file some_component.cpp
```

(continues on next page)

(continued from previous page)

```
#include "some_component.hpp"

// The following code generates all necessary boiler plate to enable the
// remote creation of 'app::some_component' instances with 'hpx::new_<>()'
//
using some_component = app::some_component;
using some_component_type = hpx::components::component<some_component>;

// Please note that the second argument to this macro must be a
// (system-wide) unique C++-style identifier (without any namespaces)
//
HPX_REGISTER_COMPONENT(some_component_type, some_component);

// The parameters for this macro have to be the same as used in the corresponding
// HPX_REGISTER_ACTION_DECLARATION() macro invocation in the corresponding
// header file.
//
// Please note that the second argument to this macro must be a
// (system-wide) unique C++-style identifier (without any namespaces)
//
HPX_REGISTER_ACTION(app::some_component::some_member_action, some_component_some_action);
```

Defining client side representation classes

Often it is very convenient to define a separate type for a component which can be used on the client side (from where the component is instantiated and used). This step might seem as unnecessary duplicating code, however it significantly increases the type safety of the code.

A possible implementation of such a client side representation for the component described in the previous section could look like:

```
#include <hpx/include/components.hpp>

namespace app
{
    // Define a client side representation type for the component type
    // 'some_component' defined in the previous section.
    //
    struct some_component_client
      : hpx::components::client_base<some_component_client, some_component>
    {
        using base_type = hpx::components::client_base<
                some_component_client, some_component>;

        some_component_client(hpx::future<hpx::id_type> && id)
          : base_type(std::move(id))
        {}

        hpx::future<int> some_member_function(std::string const& s)
        {
            some_component::some_member_action act;
```

(continues on next page)

(continued from previous page)

```
            return hpx::async(act, get_id(), s);
        }
    };
}
```

A client side object stores the global id of the component instance it represents. This global id is accessible by calling the function `client_base<>::get_id()`. The special constructor which is provided in the example allows to create this client side object directly using the API function `hpx::new_`.

Creating component instances

Instances of defined component types can be created in two different ways. If the component to create has a defined client side representation type, then this can be used, otherwise use the server type.

The following examples assume that `some_component_type` is the type of the server side implementation of the component to create. All additional arguments (see `, ...` notation below) are passed through to the corresponding constructor calls of those objects:

```
// create one instance on the given locality
hpx::id_type here = hpx::find_here();
hpx::future<hpx::id_type> f =
    hpx::new_<some_component_type>(here, ...);

// create one instance using the given distribution
// policy (here: hpx::colocating_distribution_policy)
hpx::id_type here = hpx::find_here();
hpx::future<hpx::id_type> f =
    hpx::new_<some_component_type>(hpx::colocated(here), ...);

// create multiple instances on the given locality
hpx::id_type here = find_here();
hpx::future<std::vector<hpx::id_type>> f =
    hpx::new_<some_component_type[]>(here, num, ...);

// create multiple instances using the given distribution
// policy (here: hpx::binpacking_distribution_policy)
hpx::future<std::vector<hpx::id_type>> f = hpx::new_<some_component_type[]>(
    hpx::binpacking(hpx::find_all_localities()), num, ...);
```

The examples below demonstrate the use of the same API functions for creating client side representation objects (instead of just plain ids). These examples assume that `client_type` is the type of the client side representation of the component type to create. As above, all additional arguments (see `, ...` notation below) are passed through to the corresponding constructor calls of the server side implementation objects corresponding to the `client_type`:

```
// create one instance on the given locality
hpx::id_type here = hpx::find_here();
client_type c = hpx::new_<client_type>(here, ...);

// create one instance using the given distribution
// policy (here: hpx::colocating_distribution_policy)
hpx::id_type here = hpx::find_here();
client_type c = hpx::new_<client_type>(hpx::colocated(here), ...);
```

(continues on next page)

(continued from previous page)

```
// create multiple instances on the given locality
hpx::id_type here = hpx::find_here();
hpx::future<std::vector<client_type>> f =
    hpx::new_<client_type[]>(here, num, ...);

// create multiple instances using the given distribution
// policy (here: hpx::binpacking_distribution_policy)
hpx::future<std::vector<client_type>> f = hpx::new_<client_type[]>(
    hpx::binpacking(hpx::find_all_localities()), num, ...);
```

Using component instances

After having created the component instances as described above, we can simply use them as indicated below:

```
#include <hpx/include/components.hpp>
#include <iostream>
#include <vector>

// Define a simple component
struct some_component : hpx::components::component_base<some_component>
{
    void print() const
    {
        std::cout << "Hello from component instance!" << std::endl;
    }
    HPX_DEFINE_COMPONENT_ACTION(some_component, print, print_action);
};

typedef some_component::print_action print_action;

// Create one instance on the given locality
hpx::id_type here = hpx::find_here();
hpx::future<hpx::id_type> f1 =
    hpx::new_<some_component>(here);

// Get the future value
hpx::id_type instance_id = f1.get();

// Invoke action on the instance
hpx::async<print_action>(instance_id).get();

// Create multiple instances on the given locality
int num = 3;
hpx::future<std::vector<hpx::id_type>> f2 =
    hpx::new_<some_component[]>(here, num);

// Get the future value
std::vector<hpx::id_type> instance_ids = f2.get();

// Invoke action on each instance
```

(continues on next page)

(continued from previous page)

```
for (const auto& id : instance_ids)
{
    hpx::async<print_action>(id).get();
}
```

We can use the component instances with distribution policies the same way.

Segmented containers

In parallel programming, there is now a plethora of solutions aimed at implementing "partially contiguous" or segmented data structures, whether on shared memory systems or distributed memory systems. *HPX* implements such structures by drawing inspiration from Standard C++ containers.

Using segmented containers

A segmented container is a template class that is described in the namespace `hpx`. All segmented containers are very similar semantically to their sequential counterpart (defined in `namespace std` but with an additional template parameter named `DistPolicy`). The distribution policy is an optional parameter that is passed last to the segmented container constructor (after the container size when no default value is given, after the default value if not). The distribution policy describes the manner in which a container is segmented and the placement of each segment among the available runtime localities.

However, only a part of the `std` container member functions were reimplemented:

- `(constructor)`, `(destructor)`, `operator=`
- `operator[]`
- `begin`, `cbegin`, `end`, `cend`
- `size`

An example of how to use the `partitioned_vector` container would be:

```
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

// By default, the number of segments is equal to the current number of
// localities
//
hpx::partitioned_vector<double> va(50);
hpx::partitioned_vector<double> vb(50, 0.0);
```

An example of how to use the `partitioned_vector` container with distribution policies would be:

```
#include <hpx/include/partitioned_vector.hpp>
#include <hpx/runtime_distributed/find_localities.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
```

(continues on next page)

(continued from previous page)

```
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

std::size_t num_segments = 10;
std::vector<hpx::id_type> locs = hpx::find_all_localities()

auto layout =
        hpx::container_layout( num_segments, locs );

// The number of segments is 10 and those segments are spread across the
// localities collected in the variable locs in a Round-Robin manner
//
hpx::partitioned_vector<double> va(50, layout);
hpx::partitioned_vector<double> vb(50, 0.0, layout);
```

By definition, a segmented container must be accessible from any thread although its construction is synchronous only for the thread who has called its constructor. To overcome this problem, it is possible to assign a symbolic name to the segmented container:

```
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

hpx::future<void> fserver = hpx::async(
  [](){
    hpx::partitioned_vector<double> v(50);

    // Register the 'partitioned_vector' with the name "some_name"
    //
    v.register_as("some_name");

    /* Do some code  */
  });

hpx::future<void> fclient =
  hpx::async(
    [](){
      // Naked 'partitioned_vector'
      //
      hpx::partitioned_vector<double> v;

      // Now the variable v points to the same 'partitioned_vector' that has
      // been registered with the name "some_name"
      //
      v.connect_to("some_name");

      /* Do some code  */
    });
```

Segmented containers

HPX provides the following segmented containers:

Table 2.23: Sequence containers

Name	Description	In header	C++ standard
hpx::partitioned_vector	Dynamic segmented contiguous array.	<hpx/include/partitioned_vector.hpp>	vector[160]

Table 2.24: Unordered associative containers

Name	Description	In header	C++ standard
hpx::unordered_map	Segmented collection of key-value pairs, hashed by keys, keys are unique.	<hpx/include/unordered_map.hpp>	unordered_map[161]

Segmented iterators and segmented iterator traits

The basic iterator used in the STL library is only suitable for one-dimensional structures. The iterators we use in *HPX* must adapt to the segmented format of our containers. Our iterators are then able to know when incrementing themselves if the next element of type T is in the same data segment or in another segment. In this second case, the iterator will automatically point to the beginning of the next segment.

Note: Note that the dereference operation `operator *` does not directly return a reference of type `T&` but an intermediate object wrapping this reference. When this object is used as an l-value, a remote write operation is performed; When this object is used as an r-value, implicit conversion to `T` type will take care of performing remote read operation.

It is sometimes useful not only to iterate element by element, but also segment by segment, or simply get a local iterator in order to avoid additional construction costs at each deferencing operations. To mitigate this need, the `hpx::traits::segmented_iterator_traits` are used.

With `segmented_iterator_traits` users can uniformly get the iterators which specifically iterates over segments (by providing a segmented iterator as a parameter), or get the local begin/end iterators of the nearest local segment (by providing a per-segment iterator as a parameter):

```
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

using iterator = hpx::partitioned_vector<T>::iterator;
using traits   = hpx::traits::segmented_iterator_traits<iterator>;

hpx::partitioned_vector<T> v;
std::size_t count = 0;
```

(continues on next page)

160 http://en.cppreference.com/w/cpp/container/vector
161 http://en.cppreference.com/w/cpp/container/unordered_map

(continued from previous page)

```
auto seg_begin = traits::segment(v.begin());
auto seg_end   = traits::segment(v.end());

// Iterate over segments
for (auto seg_it = seg_begin; seg_it != seg_end; ++seg_it)
{
    auto loc_begin = traits::begin(seg_it);
    auto loc_end   = traits::end(seg_it);

    // Iterate over elements inside segments
    for (auto lit = loc_begin; lit != loc_end; ++lit, ++count)
    {
        *lit = count;
    }
}
```

Which is equivalent to:

```
hpx::partitioned_vector<T> v;
std::size_t count = 0;

auto begin = v.begin();
auto end   = v.end();

for (auto it = begin; it != end; ++it, ++count)
{
    *it = count;
}
```

Using views

The use of multidimensional arrays is quite common in the numerical field whether to perform dense matrix operations or to process images. It exist many libraries which implement such object classes overloading their basic operators (e.g. +, -, *, (), etc.). However, such operation becomes more delicate when the underlying data layout is segmented or when it is mandatory to use optimized linear algebra subroutines (i.e. BLAS subroutines).

Our solution is thus to relax the level of abstraction by allowing the user to work not directly on n-dimensionnal data, but on "n-dimensionnal collections of 1-D arrays". The use of well-accepted techniques on contiguous data is thus preserved at the segment level, and the composability of the segments is made possible thanks to multidimensional array-inspired access mode.

Preface: Why SPMD?

Although *HPX* refutes by design this programming model, the *locality* plays a dominant role when it comes to implement vectorized code. To maximize local computations and avoid unneeded data transfers, a parallel section (or Single Programming Multiple Data section) is required. Because the use of global variables is prohibited, this parallel section is created via the RAII idiom.

To define a parallel section, simply write an action taking a `spmd_block` variable as a first parameter:

```
#include <hpx/collectives/spmd_block.hpp>

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */)
{
    // Parallel section

    /* Do some code */
}
HPX_PLAIN_ACTION(bulk_function, bulk_action);
```

Note: In the following paragraphs, we will use the term "image" several times. An image is defined as a lightweight process whose entry point is a function provided by the user. It's an "image of the function".

The `spmd_block` class contains the following methods:

- Team information: `get_num_images`, `this_image`, `images_per_locality`
- Control statements: `sync_all`, `sync_images`

Here is a sample code summarizing the features offered by the `spmd_block` class:

```
#include <hpx/collectives/spmd_block.hpp>

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */)
{
    std::size_t num_images = block.get_num_images();
    std::size_t this_image = block.this_image();
    std::size_t images_per_locality = block.images_per_locality();

    /* Do some code */

    // Synchronize all images in the team
    block.sync_all();

    /* Do some code */

    // Synchronize image 0 and image 1
    block.sync_images(0,1);

    /* Do some code */

    std::vector<std::size_t> vec_images = {2,3,4};

    // Synchronize images 2, 3 and 4
    block.sync_images(vec_images);
```

(continues on next page)

(continued from previous page)

```
    // Alternative call to synchronize images 2, 3 and 4
    block.sync_images(vec_images.begin(), vec_images.end());

    /* Do some code */

    // Non-blocking version of sync_all()
    hpx::future<void> event =
        block.sync_all(hpx::launch::async);

    // Callback waiting for 'event' to be ready before being scheduled
    hpx::future<void> cb =
        event.then(
          [](hpx::future<void>)
          {

            /* Do some code */

          });

    // Finally wait for the execution tree to be finished
    cb.get();
}
HPX_PLAIN_ACTION(bulk_test_function, bulk_test_action);
```

Then, in order to invoke the parallel section, call the function `define_spmd_block` specifying an arbitrary symbolic name and indicating the number of images per *locality* to create:

```
void bulk_function(hpx::lcos::spmd_block block, /* , arg0, arg1, ... */)
{

}
HPX_PLAIN_ACTION(bulk_test_function, bulk_test_action);

int main()
{
    /* std::size_t arg0, arg1, ...; */

    bulk_action act;
    std::size_t images_per_locality = 4;

    // Instantiate the parallel section
    hpx::lcos::define_spmd_block(
        "some_name", images_per_locality, std::move(act) /*, arg0, arg1, ... */);

    return 0;
}
```

Note: In principle, the user should never call the `spmd_block` constructor. The `define_spmd_block` function is responsible of instantiating `spmd_block` objects and broadcasting them to each created image.

SPMD multidimensional views

Some classes are defined as "container views" when the purpose is to observe and/or modify the values of a container using another perspective than the one that characterizes the container. For example, the values of an `std::vector` object can be accessed via the expression `[i]`. Container views can be used, for example, when it is desired for those values to be "viewed" as a 2D matrix that would have been flattened in a `std::vector`. The values would be possibly accessible via the expression `vv(i,j)` which would call internally the expression `v[k]`.

By default, the `partitioned_vector` class integrates 1-D views of its segments:

```
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

using iterator = hpx::partitioned_vector<double>::iterator;
using traits   = hpx::traits::segmented_iterator_traits<iterator>;

hpx::partitioned_vector<double> v;

// Create a 1-D view of the vector of segments
auto vv = traits::segment(v.begin());

// Access segment i
std::vector<double> v = vv[i];
```

Our views are called "multidimensional" in the sense that they generalize to N dimensions the purpose of `segmented_iterator_traits::segment()` in the 1-D case. Note that in a parallel section, the 2-D expression `a(i,j) = b(i,j)` is quite confusing because without convention, each of the images invoked will race to execute the statement. For this reason, our views are not only multidimensional but also "spmd-aware".

Note: SPMD-awareness: The convention is simple. If an assignment statement contains a view subscript as an l-value, it is only and only the image holding the r-value who is evaluating the statement. (In MPI sense, it is called a Put operation).

Subscript-based operations

Here are some examples of using subscripts in the 2-D view case:

```
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

using Vec = hpx::partitioned_vector<double>;
using View_2D = hpx::partitioned_vector_view<double,2>;
```

(continues on next page)

(continued from previous page)

```
/* Do some code */

Vec v;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t height, width;

    // Instantiate the view
    View_2D vv(block, v.begin(), v.end(), {height,width});

    // The l-value is a view subscript, the image that owns vv(1,0)
    // evaluates the assignment.
    vv(0,1) = vv(1,0);

    // The l-value is a view subscript, the image that owns the r-value
    // (result of expression 'std::vector<double>(4,1.0)') evaluates the
    // assignment : oops! race between all participating images.
    vv(2,3) = std::vector<double>(4,1.0);
}
```

Iterator-based operations

Here are some examples of using iterators in the 3-D view case:

```
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(int);

using Vec = hpx::partitioned_vector<int>;
using View_3D = hpx::partitioned_vector_view<int,3>;

/* Do some code */

Vec v1, v2;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t sixe_x, size_y, size_z;

    // Instantiate the views
    View_3D vv1(block, v1.begin(), v1.end(), {sixe_x,size_y,size_z});
    View_3D vv2(block, v2.begin(), v2.end(), {sixe_x,size_y,size_z});

    // Save previous segments covered by vv1 into segments covered by vv2
    auto vv2_it = vv2.begin();
```

(continues on next page)

(continued from previous page)

```
    auto vv1_it = vv1.cbegin();

    for(; vv2_it != vv2.end(); vv2_it++, vv1_it++)
    {
        // It's a Put operation
        *vv2_it = *vv1_it;
    }

    // Ensure that all images have performed their Put operations
    block.sync_all();

    // Ensure that only one image is putting updated data into the different
    // segments covered by vv1
    if(block.this_image() == 0)
    {
        int idx = 0;

        // Update all the segments covered by vv1
        for(auto i = vv1.begin(); i != vv1.end(); i++)
        {
            // It's a Put operation
            *i = std::vector<float>(elt_size,idx++);
        }
    }
}
```

Here is an example that shows how to iterate only over segments owned by the current image:

```
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/components/containers/partitioned_vector/partitioned_vector_local_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(float);

using Vec = hpx::partitioned_vector<float>;
using View_1D = hpx::partitioned_vector_view<float,1>;

/* Do some code */

Vec v;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t num_segments;

    // Instantiate the view
    View_1D vv(block, v.begin(), v.end(), {num_segments});

    // Instantiate the local view from the view
```

(continues on next page)

(continued from previous page)

```
    auto local_vv = hpx::local_view(vv);

    for ( auto i = local_vv.begin(); i != local_vv.end(); i++ )
    {
        std::vector<float> & segment = *i;

        /* Do some code */
    }

}
```

Instantiating sub-views

It is possible to construct views from other views: we call it sub-views. The constraint nevertheless for the subviews is to retain the dimension and the value type of the input view. Here is an example showing how to create a sub-view:

```
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(float);

using Vec = hpx::partitioned_vector<float>;
using View_2D = hpx::partitioned_vector_view<float,2>;

/* Do some code */

Vec v;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t N = 20;
    std::size_t tilesize = 5;

    // Instantiate the view
    View_2D vv(block, v.begin(), v.end(), {N,N});

    // Instantiate the subview
    View_2D svv(
        block,&vv(tilesize,0),&vv(2*tilesize-1,tilesize-1),{tilesize,tilesize},{N,N});

    if(block.this_image() == 0)
    {
        // Equivalent to 'vv(tilesize,0) = 2.0f'
        svv(0,0) = 2.0f;

        // Equivalent to 'vv(2*tilesize-1,tilesize-1) = 3.0f'
        svv(tilesize-1,tilesize-1) = 3.0f;
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Note: The last parameter of the subview constructor is the size of the original view. If one would like to create a subview of the subview and so on, this parameter should stay unchanged. `{N,N}` for the above example).

C++ co-arrays

Fortran has extended its scalar element indexing approach to reference each segment of a distributed array. In this extension, a segment is attributed a ?co-index? and lives in a specific *locality*. A co-index provides the application with enough information to retrieve the corresponding data reference. In C++, containers present themselves as a ?smarter? alternative of Fortran arrays but there are still no corresponding standardized features similar to the Fortran co-indexing approach. We present here an implementation of such features in *HPX*.

Preface: co-array, a segmented container tied to a SPMD multidimensional views

As mentioned before, a co-array is a distributed array whose segments are accessible through an array-inspired access mode. We have previously seen that it is possible to reproduce such access mode using the concept of views. Nevertheless, the user must pre-create a segmented container to instantiate this view. We illustrate below how a single constructor call can perform those two operations:

```
#include <hpx/components/containers/coarray/coarray.hpp>
#include <hpx/collectives/spmd_block.hpp>

// The following code generates all necessary boiler plate to enable the
// co-creation of 'coarray'
//
HPX_REGISTER_COARRAY(double);

// Parallel section (suppose 'block' an spmd_block instance)
{
    using hpx::container::placeholders::_;

    std::size_t height=32, width=4, segment_size=10;

    hpx::coarray<double,3> a(block, "a", {height,width,_}, segment_size);

    /* Do some code */
}
```

Unlike segmented containers, a co-array object can only be instantiated within a parallel section. Here is the description of the parameters to provide to the coarray constructor:

Table 2.25: Parameters of coarray constructor

Parameter	Description
`block`	Reference to a `spmd_block` object
`"a"`	Symbolic name of type `std::string`
`{height,width,_}`	Dimensions of the `coarray` object
`segment_size`	Size of a co-indexed element (i.e. size of the object referenced by the expression `a(i,j,k)`)

Note that the "last dimension size" cannot be set by the user. It only accepts the constexpr variable `hpx::container::placeholders::_`. This size, which is considered private, is equal to the number of current images (value returned by `block.get_num_images()`).

Note: An important constraint to remember about coarray objects is that all segments sharing the same "last dimension index" are located in the same image.

Using co-arrays

The member functions owned by the `coarray` objects are exactly the same as those of spmd multidimensional views. These are:

```
* Subscript-based operations
* Iterator-based operations
```

However, one additional functionality is provided. Knowing that the element `a(i,j,k)` is in the memory of the `k`th image, the use of local subscripts is possible.

Note: For spmd multidimensional views, subscripts are only global as it still involves potential remote data transfers.

Here is an example of using local subscripts:

```
#include <hpx/components/containers/coarray/coarray.hpp>
#include <hpx/collectives/spmd_block.hpp>

// The following code generates all necessary boiler plate to enable the
// co-creation of 'coarray'
//
HPX_REGISTER_COARRAY(double);

// Parallel section (suppose 'block' an spmd_block instance)
{
    using hpx::container::placeholders::_;

    std::size_t height=32, width=4, segment_size=10;

    hpx::coarray<double,3> a(block, "a", {height,width,_}, segment_size);

    double idx = block.this_image()*height*width;

    for (std::size_t j = 0; j<width; j++)
```

(continues on next page)

(continued from previous page)

```
    for (std::size_t i = 0; i<height; i++)
    {
        // Local write operation performed via the use of local subscript
        a(i,j,_) = std::vector<double>(elt_size,idx);
        idx++;
    }

    block.sync_all();
}
```

Note: When the "last dimension index" of a subscript is equal to `hpx::container::placeholders::_`, local subscript (and not global subscript) is used. It is equivalent to a global subscript used with a "last dimension index" equal to the value returned by `block.this_image()`.

2.3.12 Running on batch systems

This section walks you through launching *HPX* applications on various batch systems.

How to use *HPX* applications with PBS

Most *HPX* applications are executed on parallel computers. These platforms typically provide integrated job management services that facilitate the allocation of computing resources for each parallel program. *HPX* includes support for one of the most common job management systems, the Portable Batch System (PBS).

All PBS jobs require a script to specify the resource requirements and other parameters associated with a parallel job. The PBS script is basically a shell script with PBS directives placed within commented sections at the beginning of the file. The remaining (not commented-out) portions of the file executes just like any other regular shell script. While the description of all available PBS options is outside the scope of this tutorial (the interested reader may refer to in-depth documentation[162] for more information), below is a minimal example to illustrate the approach. The following test application will use the multithreaded `hello_world_distributed` program, explained in the section *Remote execution with actions*.

```
#!/bin/bash
#
#PBS -l nodes=2:ppn=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

pbsdsh -u $APP_PATH $APP_OPTIONS --hpx:nodes=`cat $PBS_NODEFILE`
```

Caution: If the first application specific argument (inside `$APP_OPTIONS`) is a non-option (i.e., does not start with a `-` or a `--`), then the argument has to be placed before the option *`--hpx:nodes`*, which, in this case, should be the last option on the command line.

Alternatively, use the option *`--hpx:endnodes`* to explicitly mark the end of the list of node names:

```
$ pbsdsh -u $APP_PATH --hpx:nodes`cat $PBS_NODEFILE` --hpx:endnodes $APP_OPTIONS
```

[162] http://www.clusterresources.com/torquedocs21/

The #PBS -l nodes=2:ppn=4 directive will cause two compute nodes to be allocated for the application, as specified in the option nodes. Each of the nodes will dedicate four cores to the program, as per the option ppn, short for "processors per node" (PBS does not distinguish between processors and cores). Note that requesting more cores per node than physically available is pointless and may prevent PBS from accepting the script.

On newer PBS versions the PBS command syntax might be different. For instance, the PBS script above would look like:

```
#!/bin/bash
#
#PBS -l select=2:ncpus=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

pbsdsh -u $APP_PATH $APP_OPTIONS --hpx:nodes=`cat $PBS_NODEFILE`
```

APP_PATH and APP_OPTIONS are shell variables that respectively specify the correct path to the executable (hello_world_distributed in this case) and the command line options. Since the hello_world_distributed application doesn't need any command line options, APP_OPTIONS has been left empty. Unlike in other execution environments, there is no need to use the *--hpx:threads* option to indicate the required number of OS threads per node; the *HPX* library will derive this parameter automatically from PBS.

Finally, pbsdsh is a PBS command that starts tasks to the resources allocated to the current job. It is recommended to leave this line as shown and modify only the PBS options and shell variables as needed for a specific application.

Important: A script invoked by pbsdsh starts in a very basic environment: the user's $HOME directory is defined and is the current directory, the LANG variable is set to C and the PATH is set to the basic /usr/local/bin:/usr/bin:/bin as defined in a system-wide file pbs_environment. Nothing that would normally be set up by a system shell profile or user shell profile is defined, unlike the environment for the main job script.

Another choice is for the pbsdsh command in your main job script to invoke your program via a shell, like sh or bash, so that it gives an initialized environment for each instance. Users can create a small script runme.sh, which is used to invoke the program:

```
#!/bin/bash
# Small script which invokes the program based on what was passed on its
# command line.
#
# This script is executed by the bash shell which will initialize all
# environment variables as usual.
$@
```

Now, the script is invoked using the pbsdsh tool:

```
#!/bin/bash
#
#PBS -l nodes=2:ppn=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=
```

(continues on next page)

(continued from previous page)

```
pbsdsh -u runme.sh $APP_PATH $APP_OPTIONS --hpx:nodes=`cat $PBS_NODEFILE`
```

All that remains now is submitting the job to the queuing system. Assuming that the contents of the PBS script were saved in the file `pbs_hello_world.sh` in the current directory, this is accomplished by typing:

```
$ qsub ./pbs_hello_world_pbs.sh
```

If the job is accepted, qsub will print out the assigned job ID, which may look like:

```
$ 42.supercomputer.some.university.edu
```

To check the status of your job, issue the following command:

```
$ qstat 42.supercomputer.some.university.edu
```

and look for a single-letter job status symbol. The common cases include:

- *Q* - signifies that the job is queued and awaiting its turn to be executed.
- *R* - indicates that the job is currently running.
- *C* - means that the job has completed.

The example qstat output below shows a job waiting for execution resources to become available:

```
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
42.supercomputer          ...ello_world.sh joe_user               0 Q batch
```

After the job completes, PBS will place two files, `pbs_hello_world.sh.o42` and `pbs_hello_world.sh.e42`, in the directory where the job was submitted. The first contains the standard output and the second contains the standard error from all the nodes on which the application executed. In our example, the error output file should be empty and the standard output file should contain something similar to:

```
hello world from OS-thread 3 on locality 0
hello world from OS-thread 2 on locality 0
hello world from OS-thread 1 on locality 1
hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 1
hello world from OS-thread 2 on locality 1
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 1
```

Congratulations! You have just run your first distributed *HPX* application!

How to use *HPX* applications with SLURM

Just like PBS (described in section *How to use HPX applications with PBS*), SLURM is a job management system which is widely used on large supercomputing systems. Any *HPX* application can easily be run using SLURM. This section describes how this can be done.

The easiest way to run an *HPX* application using SLURM is to utilize the command line tool srun, which interacts with the SLURM batch scheduling system:

```
$ srun -p <partition> -N <number-of-nodes> hpx-application <application-arguments>
```

Here, `<partition>` is one of the node partitions existing on the target machine (consult the machine's documentation to get a list of existing partitions) and `<number-of-nodes>` is the number of compute nodes that should be used. By default, the *HPX* application is started with one *locality* per node and uses all available cores on a node. You can change the number of localities started per node (for example, to account for NUMA effects) by specifying the `-n` option of srun. The number of cores per *locality* can be set by `-c`. The `<application-arguments>` are any application specific arguments that need to be passed on to the application.

Note: There is no need to use any of the *HPX* command line options related to the number of localities, number of threads, or related to networking ports. All of this information is automatically extracted from the SLURM environment by the *HPX* startup code.

Important: The srun documentation explicitly states: "If `-c` is specified without `-n`, as many tasks will be allocated per node as possible while satisfying the `-c` restriction. For instance on a cluster with 8 CPUs per node, a job request for 4 nodes and 3 CPUs per task may be allocated 3 or 6 CPUs per node (1 or 2 tasks per node) depending upon resource consumption by other jobs." For this reason, it's recommended to always specify `-n <number-of-instances>`, even if `<number-of-instances>` is equal to one (1).

Interactive shells

To get an interactive development shell on one of the nodes, users can issue the following command:

```
$ srun -p <node-type> -N <number-of-nodes> --pty /bin/bash -l
```

After the shell has been opened, users can run their *HPX* application. By default, it uses all available cores. Note that if you requested one node, you don't need to do `srun` again. However, if you requested more than one node, and want to run your distributed application, you can use `srun` again to start up the distributed *HPX* application. It will use the resources that have been requested for the interactive shell.

Scheduling batch jobs

The above mentioned method of running *HPX* applications is fine for development purposes. The disadvantage that comes with `srun` is that it only returns once the application is finished. This might not be appropriate for longer-running applications (for example, benchmarks or larger scale simulations). In order to cope with that limitation, users can use the sbatch command.

The `sbatch` command expects a script that it can run once the requested resources are available. In order to request resources, users need to add `#SBATCH` comments in their script or provide the necessary parameters to `sbatch` directly. The parameters are the same as with `run`. The commands you need to execute are the same you would need to start your application as if you were in an interactive shell.

2.3.13 Debugging *HPX* applications

Using a debugger with *HPX* applications

Using a debugger such as gdb with *HPX* applications is no problem. However, there are some things to keep in mind to make the experience somewhat more productive.

Call stacks in *HPX* can often be quite unwieldy as the library is heavily templated and the call stacks can be very deep. For this reason it is sometimes a good idea compile *HPX* in `RelWithDebInfo` mode, which applies some optimizations but keeps debugging symbols. This can often compress call stacks significantly. On the other hand, stepping through the code can also be more difficult because of statements being reordered and variables being optimized away. Also, note that because *HPX* implements user-space threads and context switching, call stacks may not always be complete in a debugger.

HPX launches not only worker threads but also a few helper threads. The first thread is the main thread, which typically does no work in an *HPX* application, except at startup and shutdown. If using the default settings, *HPX* will spawn six additional threads (used for service thread pools). The first worker thread is usually the eighth thread, and most user codes will be run on these worker threads. The last thread is a helper thread used for *HPX* shutdown.

Finally, since *HPX* is a multi-threaded runtime, the following gdb options can be helpful:

```
set pagination off
set non-stop on
```

Non-stop mode allows users to have a single thread stop on a breakpoint without stopping all other threads as well.

Using sanitizers with *HPX* applications

> **Warning:** Not all parts of *HPX* are sanitizer clean. This means that users may end up with false positives from *HPX* itself when using sanitizers for their applications.

To use sanitizers with *HPX*, turn on `HPX_WITH_SANITIZERS` and turn off `HPX_WITH_STACKOVERFLOW_DETECTION` during CMake[163] configuration. It's recommended to also build Boost with the same sanitizers that will be used for *HPX*. The appropriate sanitizers can then be enabled using CMake by appending `-fsanitize=address -fno-omit-frame-pointer` to `CMAKE_CXX_FLAGS` and `-fsanitize=address` to `CMAKE_EXE_LINKER_FLAGS`. Replace `address` with the sanitizer that you want to use.

Debugging applications using core files

For *HPX* to generate useful core files, *HPX* has to be compiled without signal and exception handlers *`HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL`*. If this option is not specified, the signal handlers change the application state. For example, after a segmentation fault the stack trace will show the signal handler. Similarly, unhandled exceptions are also caught by these handlers and the stack trace will not point to the location where the unhandled exception was thrown.

In general, core files are a helpful tool to inspect the state of the application at the moment of the crash (post-mortem debugging), without the need of attaching a debugger beforehand. This approach to debugging is especially useful if the error cannot be reliably reproduced, as only a single crashed application run is required to gain potentially helpful information like a stacktrace.

To debug with core files, the operating system first has to be told to actually write them. On most Unix systems this can be done by calling:

[163] https://www.cmake.org

```
$ ulimit -c unlimited
```

in the shell. Now the debugger can be started up with:

```
$ gdb <application> <core file name>
```

The debugger should now display the last state of the application. The default file name for core files is `core`.

2.3.14 Optimizing *HPX* applications

Performance counters

Performance counters in *HPX* are used to provide information as to how well the runtime system or an application is performing. The counter data can help determine system bottlenecks, and fine-tune system and application performance. The *HPX* runtime system, its networking, and other layers provide counter data that an application can consume to provide users with information about how well the application is performing.

Applications can also use counter data to determine how much system resources to consume. For example, an application that transfers data over the network could consume counter data from a network switch to determine how much data to transfer without competing for network bandwidth with other network traffic. The application could use the counter data to adjust its transfer rate as the bandwidth usage from other network traffic increases or decreases.

Performance counters are *HPX* parallel processes that expose a predefined interface. *HPX* exposes special API functions that allow one to create, manage, and read the counter data, and release instances of performance counters. Performance Counter instances are accessed by name, and these names have a predefined structure which is described in the section *Performance counter names*. The advantage of this is that any Performance Counter can be accessed remotely (from a different *locality*) or locally (from the same *locality*). Moreover, since all counters expose their data using the same API, any code consuming counter data can be utilized to access arbitrary system information with minimal effort.

Counter data may be accessed in real time. More information about how to consume counter data can be found in the section *Consuming performance counter data*.

All *HPX* applications provide command line options related to performance counters, such as the ability to list available counter types, or periodically query specific counters to be printed to the screen or save them in a file. For more information, please refer to the section *HPX Command Line Options*.

Performance counter names

All Performance Counter instances have a name uniquely identifying each instance. This name can be used to access the counter, retrieve all related meta data, and to query the counter data (as described in the section *Consuming performance counter data*). Counter names are strings with a predefined structure. The general form of a countername is:

```
/objectname{full_instancename}/countername@parameters
```

where `full_instancename` could be either another (full) counter name or a string formatted as:

```
parentinstancename#parentindex/instancename#instanceindex
```

Each separate part of a countername (e.g., `objectname`, `countername` `parentinstancename`, `instancename`, and `parameters`) should start with a letter (`'a'`…`'z'`, `'A'`…`'Z'`) or an underscore character (`'_'`), optionally followed by letters, digits (`'0'`…`'9'`), hyphen (`'-'`), or underscore characters. Whitespace is not allowed inside a counter name. The characters `'/'`, `'{'`, `'}'`, `'#'` and `'@'` have a special meaning and are used to delimit the different parts of the counter name.

The parts `parentinstanceindex` and `instanceindex` are integers. If an index is not specified, *HPX* will assume a default of `-1`.

Two counter name examples

This section gives examples of both simple counter names and aggregate counter names. For more information on simple and aggregate counter names, please see *Performance counter instances*.

An example of a well-formed (and meaningful) simple counter name would be:

```
/threads{locality#0/total}/count/cumulative
```

This counter returns the current cumulative number of executed (retired) *HPX* threads for the *locality* `0`. The counter type of this counter is `/threads/count/cumulative` and the full instance name is `locality#0/total`. This counter type does not require an `instanceindex` or `parameters` to be specified.

In this case, the `parentindex` (the `'0'`) designates the *locality* for which the counter instance is created. The counter will return the number of *HPX* threads retired on that particular *locality*.

Another example for a well formed (aggregate) counter name is:

```
/statistics{/threads{locality#0/total}/count/cumulative}/average@500
```

This counter takes the simple counter from the first example, samples its values every `500` milliseconds, and returns the average of the value samples whenever it is queried. The counter type of this counter is `/statistics/average` and the instance name is the full name of the counter for which the values have to be averaged. In this case, the `parameters` (the `'500'`) specify the sampling interval for the averaging to take place (in milliseconds).

Performance counter types

Every performance counter belongs to a specific performance counter type which classifies the counters into groups of common semantics. The type of a counter is identified by the `objectname` and the `countername` parts of the name.

```
/objectname/countername
```

When an application starts *HPX* will register all available counter types on each of the localities. These counter types are held in a special performance counter registration database, which can be used to retrieve the meta data related to a counter type and to create counter instances based on a given counter instance name.

Performance counter instances

The `full_instancename` distinguishes different counter instances of the same counter type. The formatting of the `full_instancename` depends on the counter type. There are two types of counters: simple counters, which usually generate the counter values based on direct measurements, and aggregate counters, which take another counter and transform its values before generating their own counter values. An example for a simple counter is given *above*: counting retired *HPX* threads. An aggregate counter is shown as an example *above* as well: calculating the average of the underlying counter values sampled at constant time intervals.

While simple counters use instance names formatted as `parentinstancename#parentindex/instancename#instanceindex`, most aggregate counters have the full counter name of the embedded counter as their instance name.

Not all simple counter types require specifying all four elements of a full counter instance name; some of the parts (`parentinstancename`, `parentindex`, `instancename`, and `instanceindex`) are optional for specific counters.

Please refer to the documentation of a particular counter for more information about the formatting requirements for the name of this counter (see *Existing HPX performance counters*).

The `parameters` are used to pass additional information to a counter at creation time. They are optional, and they fully depend on the concrete counter. Even if a specific counter type allows additional parameters to be given, those usually are not required as sensible defaults will be chosen. Please refer to the documentation of a particular counter for more information about what parameters are supported, how to specify them, and what default values are assumed (see also *Existing HPX performance counters*).

Every *locality* of an application exposes its own set of performance counter types and performance counter instances. The set of exposed counters is determined dynamically at application start based on the execution environment of the application. For instance, this set is influenced by the current hardware environment for the *locality* (such as whether the *locality* has access to accelerators), and the software environment of the application (such as the number of OS threads used to execute *HPX* threads).

Using wildcards in performance counter names

It is possible to use wildcard characters when specifying performance counter names. Performance counter names can contain two types of wildcard characters:

- Wildcard characters in the performance counter type
- Wildcard characters in the performance counter instance name

A wildcard character has a meaning which is very close to usual file name wildcard matching rules implemented by common shells (like bash).

Table 2.26: Wildcard characters in the performance counter type

Wildcard	Description
`*`	This wildcard character matches any number (zero or more) of arbitrary characters.
`?`	This wildcard character matches any single arbitrary character.
`[...]`	This wildcard character matches any single character from the list of specified within the square brackets.

Table 2.27: Wildcard characters in the performance counter instance name

Wildcard	Description
`*`	This wildcard character matches any *locality* or any thread, depending on whether it is used for `locality#*` or `worker-thread#*`. No other wildcards are allowed in counter instance names.

Consuming performance counter data

You can consume performance data using either the command line interface, the *HPX* application or the *HPX* API. The command line interface is easier to use, but it is less flexible and does not allow one to adjust the behaviour of your application at runtime. The command line interface provides a convenience abstraction but simplified abstraction for querying and logging performance counter data for a set of performance counters.

Consuming performance counter data from the command line

HPX provides a set of predefined command line options for every application that uses `hpx::init` for its initialization. While there are many more command line options available (see *HPX Command Line Options*), the set of options related to performance counters allows one to list existing counters, and query existing counters once at application termination or repeatedly after a constant time interval.

The following table summarizes the available command line options:

Table 2.28: *HPX* Command Line Options Related to Performance Counters

Command line option	Description
`--hpx:print-counter`	Prints the specified performance counter either repeatedly and/or at the times specified by `--hpx:print-counter-at` (see also option `--hpx:print-counter-interval`).
`--hpx:print-counter-reset`	Prints the specified performance counter either repeatedly and/or at the times specified by `--hpx:print-counter-at`. Reset the counter after the value is queried (see also option `--hpx:print-counter-interval`).
`--hpx:print-counter-interval`	Prints the performance counter(s) specified with `--hpx:print-counter` repeatedly after the time interval (specified in milliseconds) (default:`0` which means print once at shutdown).
`--hpx:print-counter-destination`	Prints the performance counter(s) specified with `--hpx:print-counter` to the given file (default: console).
`--hpx:list-counters`	Lists the names of all registered performance counters.
`--hpx:list-counter-infos`	Lists the description of all registered performance counters.
`--hpx:print-counter-format`	Prints the performance counter(s) specified with `--hpx:print-counter`. Possible formats in CVS format with header or without any header (see option `--hpx:no-csv-header`), possible values: `csv` (prints counter values in CSV format with full names as header) `csv-short` (prints counter values in CSV format with shortnames provided with `--hpx:print-counter` as `--hpx:print-counter shortname,full-countername`).
`--hpx:no-csv-header`	Prints the performance counter(s) specified with `--hpx:print-counter` and `csv` or `csv-short` format specified with `--hpx:print-counter-format` without header.
`--hpx:print-counter-at arg`	Prints the performance counter(s) specified with `--hpx:print-counter` (or `--hpx:print-counter-reset`) at the given point in time. Possible argument values: `startup`, `shutdown` (default), `noshutdown`.
`--hpx:reset-counters`	Resets all performance counter(s) specified with `--hpx:print-counter` after they have been evaluated.
`--hpx:print-counter-types`	Appends counter type description to generated output.
`--hpx:print-counters-locally`	Each locality prints only its own local counters.

While the options `--hpx:list-counters` and `--hpx:list-counter-infos` give a short list of all available counters, the full documentation for those can be found in the section *Existing HPX performance counters*.

A simple example

All of the commandline options mentioned above can be tested using the `hello_world_distributed` example.

Listing all available counters `hello_world_distributed --hpx:list-counters` yields:

```
List of available counter instances (replace * below with the appropriate
sequence number)
-------------------------------------------------------------------------
/agas/count/allocate /agas/count/bind /agas/count/bind_gid
/agas/count/bind_name ... /threads{locality#*/allocator#*}/count/objects
/threads{locality#*/total}/count/stack-recycles
/threads{locality#*/total}/idle-rate
/threads{locality#*/worker-thread#*}/idle-rate
```

Providing more information about all available counters, `hello_world_distributed --hpx:list-counter-infos` yields:

```
Information about available counter instances (replace * below with the
appropriate sequence number)
------------------------------------------------------------------------------
fullname: /agas/count/allocate helptext: returns the number of invocations of
the AGAS service 'allocate' type: counter_type::raw version: 1.0.0
------------------------------------------------------------------------------

------------------------------------------------------------------------------
fullname: /agas/count/bind helptext: returns the number of invocations of the
AGAS service 'bind' type: counter_type::raw version: 1.0.0
------------------------------------------------------------------------------

------------------------------------------------------------------------------
fullname: /agas/count/bind_gid helptext: returns the number of invocations of
the AGAS service 'bind_gid' type: counter_type::raw version: 1.0.0
------------------------------------------------------------------------------

...
```

This command will not only list the counter names but also a short description of the data exposed by this counter.

Note: The list of available counters may differ depending on the concrete execution environment (hardware or software) of your application.

Requesting the counter data for one or more performance counters can be achieved by invoking `hello_world_distributed` with a list of counter names:

```
$ hello_world_distributed \
    --hpx:print-counter=/threads{locality#0/total}/count/cumulative \
    --hpx:print-counter=/agas{locality#0/total}/count/bind
```

which yields for instance:

```
hello world from OS-thread 0 on locality 0
/threads{locality#0/total}/count/cumulative,1,0.212527,[s],33
/agas{locality#0/total}/count/bind,1,0.212790,[s],11
```

The first line is the normal output generated by `hello_world_distributed` and has no relation to the counter data listed. The last two lines contain the counter data as gathered at application shutdown. These lines have six fields, the counter name, the sequence number of the counter invocation, the time stamp at which this information has been sampled, the unit of measure for the time stamp, the actual counter value and an optional unit of measure for the counter value.

Note: The command line option `--hpx:print-counter-types` will append a seventh field to the generated output. This field will hold an abbreviated counter type.

The actual counter value can be represented by a single number (for counters returning singular values) or a list of numbers separated by `':'` (for counters returning an array of values, like for instance a histogram).

Note: The name of the performance counter will be enclosed in double quotes `'"'` if it contains one or more commas `','`.

Requesting to query the counter data once after a constant time interval with this command line:

```
$ hello_world_distributed \
    --hpx:print-counter=/threads{locality#0/total}/count/cumulative \
    --hpx:print-counter=/agas{locality#0/total}/count/bind \
    --hpx:print-counter-interval=20
```

yields for instance (leaving off the actual console output of the `hello_world_distributed` example for brevity):

```
threads{locality#0/total}/count/cumulative,1,0.002409,[s],22
agas{locality#0/total}/count/bind,1,0.002542,[s],9
threads{locality#0/total}/count/cumulative,2,0.023002,[s],41
agas{locality#0/total}/count/bind,2,0.023557,[s],10
threads{locality#0/total}/count/cumulative,3,0.037514,[s],46
agas{locality#0/total}/count/bind,3,0.038679,[s],10
```

The command `--hpx:print-counter-destination=<file>` will redirect all counter data gathered to the specified file name, which avoids cluttering the console output of your application.

The command line option `--hpx:print-counter` supports using a limited set of wildcards for a (very limited) set of use cases. In particular, all occurrences of `#*` as in `locality#*` and in `worker-thread#*` will be automatically expanded to the proper set of performance counter names representing the actual environment for the executed program. For instance, if your program is utilizing four worker threads for the execution of *HPX* threads (see command line option *`--hpx:threads`*) the following command line

```
$ hello_world_distributed \
    --hpx:threads=4 \
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative
```

will print the value of the performance counters monitoring each of the worker threads:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 0
hello world from OS-thread 2 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.0025214,[s],27
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.0025453,[s],33
```

(continues on next page)

(continued from previous page)

```
/threads{locality#0/worker-thread#2}/count/cumulative,1,0.0025683,[s],29
/threads{locality#0/worker-thread#3}/count/cumulative,1,0.0025904,[s],33
```

The command `--hpx:print-counter-format` takes values `csv` and `csv-short` to generate CSV formatted counter values with a header.

With format as csv:

```
$ hello_world_distributed \
    --hpx:threads=2 \
    --hpx:print-counter-format csv \
    --hpx:print-counter /threads{locality#*/total}/count/cumulative \
    --hpx:print-counter /threads{locality#*/total}/count/cumulative-phases
```

will print the values of performance counters in CSV format with the full countername as a header:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
/threads{locality#*/total}/count/cumulative,/threads{locality#*/total}/count/cumulative-
↪phases
39,93
```

With format csv-short:

```
$ hello_world_distributed \
    --hpx:threads 2 \
    --hpx:print-counter-format csv-short \
    --hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \
    --hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases
```

will print the values of performance counters in CSV format with the short countername as a header:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
cumulative,phases
39,93
```

With format csv and csv-short when used with `--hpx:print-counter-interval`:

```
$ hello_world_distributed \
    --hpx:threads 2 \
    --hpx:print-counter-format csv-short \
    --hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \
    --hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases \
    --hpx:print-counter-interval 5
```

will print the header only once repeating the performance counter value(s) repeatedly:

```
cum,phases
25,42
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
44,95
```

The command `--hpx:no-csv-header` can be used with `--hpx:print-counter-format` to print performance counter values in CSV format without any header:

```
$ hello_world_distributed \
--hpx:threads 2 \
--hpx:print-counter-format csv-short \
--hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \
--hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases \
--hpx:no-csv-header
```

will print:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
37,91
```

Consuming performance counter data using the *HPX* API

HPX provides an API that allows users to discover performance counters and to retrieve the current value of any existing performance counter from any application.

Discover existing performance counters

Retrieve the current value of any performance counter

Performance counters are specialized *HPX* components. In order to retrieve a counter value, the performance counter needs to be instantiated. *HPX* exposes a client component object for this purpose:

```
hpx::performance_counters::performance_counter counter(std::string const& name);
```

Instantiating an instance of this type will create the performance counter identified by the given `name`. Only the first invocation for any given counter name will create a new instance of that counter. All following invocations for a given counter name will reference the initially created instance. This ensures that at any point in time there is never more than one active instance of any of the existing performance counters.

In order to access the counter value (or to invoke any of the other functionality related to a performance counter, like `start`, `stop` or `reset`) member functions of the created client component instance should be called:

```
// print the current number of threads created on locality 0
hpx::performance_counters::performance_counter count(
    "/threads{locality#0/total}/count/cumulative");
hpx::cout << count.get_value<int>().get() << std::endl;
```

For more information about the client component type, see `hpx::performance_counters::performance_counter`

Note: In the above example `count.get_value()` returns a future. In order to print the result we must append `.get()` to retrieve the value. You could write the above example like this for more clarity:

```
// print the current number of threads created on locality 0
hpx::performance_counters::performance_counter count(
    "/threads{locality#0/total}/count/cumulative");
```

(continues on next page)

(continued from previous page)

```
hpx::future<int> result = count.get_value<int>();
hpx::cout << result.get() << std::endl;
```

Providing performance counter data

HPX offers several ways by which you may provide your own data as a performance counter. This has the benefit of exposing additional, possibly application-specific information using the existing Performance Counter framework, unifying the process of gathering data about your application.

An application that wants to provide counter data can implement a performance counter to provide the data. When a consumer queries performance data, the *HPX* runtime system calls the provider to collect the data. The runtime system uses an internal registry to determine which provider to call.

Generally, there are two ways of exposing your own performance counter data: a simple, function-based way and a more complex, but more powerful way of implementing a full performance counter. Both alternatives are described in the following sections.

Exposing performance counter data using a simple function

The simplest way to expose arbitrary numeric data is to write a function which will then be called whenever a consumer queries this counter. Currently, this type of performance counter can only be used to expose integer values. The expected signature of this function is:

```
std::int64_t some_performance_data(bool reset);
```

The argument `bool reset` (which is supplied by the runtime system when the function is invoked) specifies whether the counter value should be reset after evaluating the current value (if applicable).

For instance, here is such a function returning how often it was invoked:

```
// The atomic variable 'counter' ensures the thread safety of the counter.
boost::atomic<std::int64_t> counter(0);

std::int64_t some_performance_data(bool reset)
{
    std::int64_t result = ++counter;
    if (reset)
        counter = 0;
    return result;
}
```

This example function exposes a linearly-increasing value as our performance data. The value is incremented on each invocation, i.e., each time a consumer requests the counter data of this performance counter.

The next step in exposing this counter to the runtime system is to register the function as a new raw counter type using the *HPX* API function *hpx::performance_counters::install_counter_type*. A counter type represents certain common characteristics of counters, like their counter type name and any associated description information. The following snippet shows an example of how to register the function `some_performance_data`, which is shown above, for a counter type named `"/test/data"`. This registration has to be executed before any consumer instantiates, and queries an instance of this counter type:

```
#include <hpx/include/performance_counters.hpp>

void register_counter_type()
{
    // Call the HPX API function to register the counter type.
    hpx::performance_counters::install_counter_type(
        "/test/data",                                  // counter type name
        &some_performance_data,                        // function providing counter␣
↪data
        "returns a linearly increasing counter value"  // description text (optional)
        ""                                             // unit of measure (optional)
    );
}
```

Now it is possible to instantiate a new counter instance based on the naming scheme `"/test{locality#*/total}/data"` where `*` is a zero-based integer index identifying the *locality* for which the counter instance should be accessed. The function `hpx::performance_counters::install_counter_type` enables users to instantiate exactly one counter instance for each *locality*. Repeated requests to instantiate such a counter will return the same instance, i.e., the instance created for the first request.

If this counter needs to be accessed using the standard *HPX* command line options, the registration has to be performed during application startup, before `hpx_main` is executed. The best way to achieve this is to register an *HPX* startup function using the API function `hpx::register_startup_function` before calling `hpx::init` to initialize the runtime system:

```
int main(int argc, char* argv[])
{
    // By registering the counter type we make it available to any consumer
    // who creates and queries an instance of the type "/test/data".
    //
    // This registration should be performed during startup. The
    // function 'register_counter_type' should be executed as an HPX thread right
    // before hpx_main is executed.
    hpx::register_startup_function(&register_counter_type);

    // Initialize and run HPX.
    return hpx::init(argc, argv);
}
```

Please see the code in `simplest_performance_counter.cpp` for a full example demonstrating this functionality.

Implementing a full performance counter

Sometimes, the simple way of exposing a single value as a performance counter is not sufficient. For that reason, *HPX* provides a means of implementing full performance counters which support:

- Retrieving the descriptive information about the performance counter
- Retrieving the current counter value
- Resetting the performance counter (value)
- Starting the performance counter
- Stopping the performance counter

- Setting the (initial) value of the performance counter

Every full performance counter will implement a predefined interface:

```
//  Copyright (c) 2007-2023 Hartmut Kaiser
//
//  SPDX-License-Identifier: BSL-1.0
//  Distributed under the Boost Software License, Version 1.0. (See accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#include <hpx/async_base/launch_policy.hpp>
#include <hpx/components/client_base.hpp>
#include <hpx/functional/bind_front.hpp>
#include <hpx/futures/future.hpp>
#include <hpx/modules/execution.hpp>

#include <hpx/performance_counters/counters_fwd.hpp>
#include <hpx/performance_counters/server/base_performance_counter.hpp>

#include <string>
#include <utility>
#include <vector>

#include <hpx/config/warnings_prefix.hpp>

///////////////////////////////////////////////////////////////////////////////
namespace hpx::performance_counters {

    ///////////////////////////////////////////////////////////////////////////
    struct HPX_EXPORT performance_counter
      : components::client_base<performance_counter,
            server::base_performance_counter>
    {
        using base_type = components::client_base<performance_counter,
            server::base_performance_counter>;

        performance_counter() = default;

        explicit performance_counter(std::string const& name);

        performance_counter(
            std::string const& name, hpx::id_type const& locality);

        performance_counter(id_type const& id)
          : base_type(id)
        {
        }

        performance_counter(future<id_type>&& id)
          : base_type(HPX_MOVE(id))
        {
```

(continues on next page)

(continued from previous page)

```
    }

    performance_counter(hpx::future<performance_counter>&& c)
      : base_type(HPX_MOVE(c))
    {
    }

    ///////////////////////////////////////////////////////////////////////
    future<counter_info> get_info() const;
    counter_info get_info(
        launch::sync_policy, error_code& ec = throws) const;

    future<counter_value> get_counter_value(bool reset) const;
    counter_value get_counter_value(
        launch::sync_policy, bool reset, error_code& ec = throws) const;

    future<counter_value> get_counter_value() const;
    counter_value get_counter_value(
        launch::sync_policy, error_code& ec = throws) const;

    future<counter_values_array> get_counter_values_array(bool reset) const;
    counter_values_array get_counter_values_array(
        launch::sync_policy, bool reset, error_code& ec = throws) const;

    future<counter_values_array> get_counter_values_array() const;
    counter_values_array get_counter_values_array(
        launch::sync_policy, error_code& ec = throws) const;

    ///////////////////////////////////////////////////////////////////////
    future<bool> start() const;
    bool start(launch::sync_policy, error_code& ec = throws) const;

    future<bool> stop() const;
    bool stop(launch::sync_policy, error_code& ec = throws) const;

    future<void> reset() const;
    void reset(launch::sync_policy, error_code& ec = throws) const;

    future<void> reinit(bool reset = true) const;
    void reinit(launch::sync_policy, bool reset = true,
        error_code& ec = throws) const;

    ///////////////////////////////////////////////////////////////////////
    future<std::string> get_name() const;
    std::string get_name(
        launch::sync_policy, error_code& ec = throws) const;

private:
    template <typename T>
    static T extract_value(future<counter_value>&& value)
    {
        return value.get().get_value<T>();
```

(continues on next page)

(continued from previous page)

```
        }

    public:
        template <typename T>
        future<T> get_value(bool reset = false)
        {
            return get_counter_value(reset).then(hpx::launch::sync,
                hpx::bind_front(&performance_counter::extract_value<T>));
        }
        template <typename T>
        T get_value(
            launch::sync_policy, bool reset = false, error_code& ec = throws)
        {
            return get_counter_value(launch::sync, reset).get_value<T>(ec);
        }

        template <typename T>
        future<T> get_value() const
        {
            return get_counter_value(false).then(hpx::launch::sync,
                hpx::bind_front(&performance_counter::extract_value<T>));
        }
        template <typename T>
        T get_value(launch::sync_policy, error_code& ec = throws) const
        {
            return get_counter_value(launch::sync, false).get_value<T>(ec);
        }
    };

    // Return all counters matching the given name (with optional wild cards).
    HPX_EXPORT std::vector<performance_counter> discover_counters(
        std::string const& name, error_code& ec = throws);
}    // namespace hpx::performance_counters

#include <hpx/config/warnings_suffix.hpp>
```

In order to implement a full performance counter, you have to create an *HPX* component exposing this interface. To simplify this task, *HPX* provides a ready-made base class which handles all the boiler plate of creating a component for you. The remainder of this section will explain the process of creating a full performance counter based on the Sine example, which you can find in the directory `examples/performance_counters/sine/`.

The base class is defined in the header file `base_performance_counter.cpp` as:

```
//  Copyright (c) 2007-2023 Hartmut Kaiser
//
//  SPDX-License-Identifier: BSL-1.0
//  Distributed under the Boost Software License, Version 1.0. (See accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#include <hpx/actions_base/component_action.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <hpx/components_base/component_type.hpp>
#include <hpx/components_base/server/component_base.hpp>
#include <hpx/performance_counters/counters.hpp>
#include <hpx/performance_counters/server/base_performance_counter.hpp>
#include <hpx/runtime_local/get_locality_id.hpp>

///////////////////////////////////////////////////////////////////////////////
//[performance_counter_base_class
namespace hpx::performance_counters {

    template <typename Derived>
    class base_performance_counter;
}    // namespace hpx::performance_counters
//]

///////////////////////////////////////////////////////////////////////////////
namespace hpx::performance_counters {

    template <typename Derived>
    class base_performance_counter
      : public hpx::performance_counters::server::base_performance_counter
      , public hpx::components::component_base<Derived>
    {
    private:
        using base_type = hpx::components::component_base<Derived>;

    public:
        using type_holder = Derived;
        using base_type_holder =
            hpx::performance_counters::server::base_performance_counter;

        base_performance_counter() = default;

        explicit base_performance_counter(
            hpx::performance_counters::counter_info const& info)
          : base_type_holder(info)
        {
        }

        // Disambiguate finalize() which is implemented in both base classes
        void finalize()
        {
            base_type_holder::finalize();
            base_type::finalize();
        }

        hpx::naming::address get_current_address() const
        {
            return hpx::naming::address(
                hpx::naming::get_gid_from_locality_id(hpx::get_locality_id()),
                hpx::components::get_component_type<Derived>(),
                const_cast<Derived*>(static_cast<Derived const*>(this)));
```

(continues on next page)

(continued from previous page)

```
        }
    };
}    // namespace hpx::performance_counters
```

The single template parameter is expected to receive the type of the derived class implementing the performance counter. In the Sine example this looks like:

```
//  Copyright (c) 2007-2012 Hartmut Kaiser
//
//  SPDX-License-Identifier: BSL-1.0
//  Distributed under the Boost Software License, Version 1.0. (See accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/hpx.hpp>
#include <hpx/include/lcos_local.hpp>
#include <hpx/include/performance_counters.hpp>
#include <hpx/include/util.hpp>

#include <cstdint>

namespace performance_counters { namespace sine { namespace server {
    ///////////////////////////////////////////////////////////////////////////
    //[sine_counter_definition
    class sine_counter
      : public hpx::performance_counters::base_performance_counter<sine_counter>
    //]
    {
    public:
        sine_counter()
          : current_value_(0)
          , evaluated_at_(0)
        {
        }
        explicit sine_counter(
            hpx::performance_counters::counter_info const& info);

        /// This function will be called in order to query the current value of
        /// this performance counter
        hpx::performance_counters::counter_value get_counter_value(bool reset);

        /// The functions below will be called to start and stop collecting
        /// counter values from this counter.
        bool start();
        bool stop();

        /// finalize() will be called just before the instance gets destructed
        void finalize();

```

(continues on next page)

(continued from previous page)

```
    protected:
        bool evaluate();

    private:
        typedef hpx::spinlock mutex_type;

        mutable mutex_type mtx_;
        double current_value_;
        std::uint64_t evaluated_at_;

        hpx::util::interval_timer timer_;
    };
}}}    // namespace performance_counters::sine::server
#endif
```

i.e., the type `sine_counter` is derived from the base class passing the type as a template argument (please see `simplest_performance_counter.cpp` for the full source code of the counter definition). For more information about this technique (called Curiously Recurring Template Pattern - CRTP), please see for instance the corresponding Wikipedia article[164]. This base class itself is derived from the `performance_counter` interface described above.

Additionally, a full performance counter implementation not only exposes the actual value but also provides information about:

- The point in time a particular value was retrieved.
- A (sequential) invocation count.
- The actual counter value.
- An optional scaling coefficient.
- Information about the counter status.

Existing *HPX* performance counters

The *HPX* runtime system exposes a wide variety of predefined performance counters. These counters expose critical information about different modules of the runtime system. They can help determine system bottlenecks and fine-tune system and application performance.

[164] http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Table 2.29: AGAS performance counter /agas/count/<agas_service>

Counter type	/agas/count/<agas_service> where <agas_service> is one of the following: *primary namespace services*: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate, begin_migration, end_migration *component namespace services*: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type *locality namespace services*: free, localities, num_localities, num_threads, resolve_locality, resolved_localities *symbol namespace services*: bind, resolve, unbind, iterate_names, on_symbol_namespace_event
Counter instance formatting	<agas_instance>/total where <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the locality hosting the *AGAS* service). The value for * can be any *locality* id for the following <agas_service>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, bin, resolve, unbind, and iterate_names (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).
Description	Returns the total number of invocations of the specified *AGAS* service since its creation.

Table 2.30: AGAS performance counter /agas/<agas_service_category>/count

Counter type	/agas/<agas_service_category>/count where <agas_service_category> is one of the following: primary, locality, component or symbol
Counter instance formatting	<agas_instance>/total where <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the *locality* hosting the *AGAS* service). Except for <agas_service_category>, primary or symbol for which the value for * can be any *locality* id (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).
Description	Returns the overall total number of invocations of all *AGAS* services provided by the given *AGAS* service category since its creation.

Table 2.31: AGAS performance counter /agas/<agas_service_category>/count

Counter type	/agas/<agas_service_category>/count where <agas_service_category> is one of the following: primary, locality, component or symbol
Counter instance formatting	<agas_instance>/total where <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the *locality* hosting the *AGAS* service). Except for <agas_service_category>, primary or symbol for which the value for * can be any *locality* id (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).
Description	Returns the overall total number of invocations of all *AGAS* services provided by the given *AGAS* service category since its creation.

Table 2.32: AGAS performance counter agas/time/<agas_service>

Counter type	agas/time/<agas_service> where <agas_service> is one of the following: *primary namespace services*: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate begin_migration, end_migration *component namespace services*: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type *locality namespace services*: free, localities, num_localities, num_threads, resolve_locality, resolved_localities *symbol namespace services*: bind, resolve, unbind, iterate_names, on_symbol_namespace_event
Counter instance formatting	<agas_instance>/total where <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the *locality* hosting the *AGAS* service). The value for * can be any *locality* id for the following <agas_service>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, bin, resolve, unbind, and iterate_names (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).
Description	Returns the overall execution time of the specified *AGAS* service since its creation (in nanoseconds).

Table 2.33: AGAS performance counter */agas/<agas_service_category>/time`*

Counter type	/agas/<agas_service_category>/time where <agas_service_category> is one of the following: primary, locality, component or symbol
Counter instance formatting	<agas_instance>/total where <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the *locality* hosting the *AGAS* service). Except for <agas_service_category primary or symbol for which the value for * can be any *locality* id (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).
Description	Returns the overall execution time of all *AGAS* services provided by the given *AGAS* service category since its creation (in nanoseconds).

Table 2.34: AGAS performance counter /agas/count/entries

Counter type	/agas/count/entries
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the *AGAS* cache should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of cache entries resident in the *AGAS* cache of the specified *locality* (see <cache_statistics>).

Table 2.35: AGAS performance counter /agas/count/<cache_statistics>

Counter type	/agas/count/<cache_statistics> where <cache_statistics> is one of the following: cache/evictions, cache/hits, cache/insertions, cache/misses
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the *AGAS* cache should be queried. The *locality* id is a (zero based) number identifying the *locality*
Description	Returns the number of cache events (evictions, hits, inserts, and misses) in the *AGAS* cache of the specified *locality* (see <cache_statistics>).

Table 2.36: AGAS performance counter /agas/count/<full_cache_statistics>

Counter type	/agas/count/<full_cache_statistics> where <full_cache_statistics> is one of the following: cache/get_entry, cache/insert_entry, cache/update_entry, cache/erase_entry
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the *AGAS* cache should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of invocations of the specified cache API function of the *AGAS* cache.

Table 2.37: AGAS performance counter /agas/time/<full_cache_statistics>

Counter type	/agas/time/<full_cache_statistics> where <full_cache_statistics> is one of the following: cache/get_entry, cache/insert_entry, cache/update_entry, cache/erase_entry
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the *AGAS* cache should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall time spent executing of the specified API function of the *AGAS* cache.

Table 2.38: Parcel layer performance counter /data/count/<connection_type>/<operation>

Counter type	/data/count/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number of raw (uncompressed) bytes sent or received (see <operation>, e.g. sent or received) for the specified <connection_type>. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.39: Parcel layer performance counter /data/time/<connection_type>/<operation>

Counter type	/data/time/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the total transmission time should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the total time (in nanoseconds) between the start of each asynchronous transmission operation and the end of the corresponding operation for the specified <connection_type> the given *locality* (see <operation>, e.g. sent or received). The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.40: Parcel layer performance counter /serialize/count/<connection_type>/<operation>

Counter type	/serialize/count/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number of bytes transferred (see <operation>, e.g. sent or received possibly compressed) for the specified <connection_type> by the given *locality*. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.
Description	If the configure-time option -DHPX_WITH_PARCELPORT_ACTION_COUNTERS=On was specified, this counter allows one to specify an optional action name as its parameter. In this case the counter will report the number of bytes transmitted for the given action only.

Table 2.41: Parcel layer performance counter /serialize/time/<connection_type>/<operation>

Counter type	/serialize/time/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the serialization time should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall time spent performing outgoing data serialization for the specified <connection_type> on the given *locality* (see <operation>, e.g. sent or received). The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.
Parameters	If the configure-time option -DHPX_WITH_PARCELPORT_ACTION_COUNTERS=On was specified, this counter allows one to specify an optional action name as its parameter. In this case the counter will report the serialization time for the given action only.

Table 2.42: Parcel layer performance counter /parcels/count/routed

Counter type	/parcels/count/routed
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the number of routed parcels should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number of routed (outbound) parcels transferred by the given *locality*. Routed parcels are those which cannot directly be delivered to its destination as the local *AGAS* is not able to resolve the destination address. In this case a parcel is sent to the *AGAS* service component which is responsible for creating the destination GID (and is responsible for resolving the destination address). This *AGAS* service component will deliver the parcel to its final target.
Parameters	If the configure-time option -DHPX_WITH_PARCELPORT_ACTION_COUNTERS=On was specified, this counter allows one to specify an optional action name as its parameter. In this case the counter will report the number of parcels for the given action only.

Table 2.43: Parcel layer performance counter /parcels/count/<connection_type>/<operation>

Counter type	/parcels/count/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the number of parcels should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number of parcels transferred using the specified <connection_type> by the given *locality* (see operation>, e.g. sent or received. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.44: Parcel layer performance counter /messages/count/<connection_type>/<operation>

Counter type	/messages/count/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the number of messages should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number of messages[165] transferred using the specified <connection_type> by the given *locality* (see <operation>, e.g. sent or received) The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.45: Parcel layer performance counter /parcelport/count/<connection_type>/zero_copy_chunks/<operation>

Counter type	/parcelport/count/<connection_type>/zero_copy_chunks/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number of zero-copy chunks sent or received (see <operation>, e.g. sent or received) for the specified <connection_type>. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

[165] A message can potentially consist of more than one *parcel*.

Table 2.46: Parcel layer performance counter /parcelport/count-max/<connection_type>/zero_copy_chunks/<operation>

Counter type	/parcelport/count-max/<connection_type>/zero_copy_chunks/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the maximum number of zero-copy chunks sent or received per message (see <operation>, e.g. sent or received) for the specified <connection_type>. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.47: Parcel layer performance counter /parcelport/size/<connection_type>/zero_copy_chunks/<operation>

Counter type	/parcelport/size/<connection_type>/zero_copy_chunks/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall size of zero-copy chunks sent or received (see <operation>, e.g. sent or received) for the specified <connection_type>. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.48: Parcel layer performance counter /parcelport/size-max/<connection_type>/zero_copy_chunks/<operation>

Counter type	/parcelport/size-max/<connection_type>/zero_copy_chunks/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the maximum size of zero-copy chunks sent or received (see <operation>, e.g. sent or received) for the specified <connection_type>. The performance counters are available only if the compile time constant HPX_HAVE_PARCELPORT_COUNTERS was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_COUNTERS. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.49: Parcel layer performance counter /parcelport/count/<connection_type>/<cache_statistics>

Counter type	/parcelport/count/<connection_type>/<cache_statistics> where: <cache_statistics> is one of the following: cache/insertions, cache/evictions, cache/hits, cache/misses <connection_type> is one of the following: tcp, mpi
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the number of messages should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number cache events (evictions, hits, inserts, misses, and reclaims) for the connection cache of the given connection type on the given *locality* (see <cache_statistics, e.g. ache/insertions, cache/evictions, cache/hits, cache/misses or``cache/reclaims``. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default). The corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake options* for more details.

Table 2.50: Parcel layer performance counter /parcelqueue/length/<operation>

Counter type	/parcelqueue/length/<operation> where <operation> is one of the following: sent, receive
Counter instance formatting	locality#*/total where * is the *locality* id of the *locality* the *parcel* queue should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the current number of parcels stored in the *parcel* queue (see <operation> for which queue to query, e.g. sent or received).

Table 2.51: Thread manager performance counter /threads/count/cumulative

Counter type	/threads/count/cumulative
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the overall number of retired *HPX*-threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. **pool#* is defining the pool for which the current value of the** idle-loop counter should be queried for. **worker-thread#* is defining the worker thread for which the overall** number of retired *HPX*-threads should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the overall number of executed (retired) *HPX*-threads on the given *locality* since application start. If the instance name is total the counter returns the accumulated number of retired *HPX*-threads for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the overall number of retired *HPX*-threads for all worker threads separately. This counter is available only if the configuration time constant HPX_WITH_THREAD_CUMULATIVE_COUNTS is set to ON (default: ON).

Table 2.52: Thread manager performance counter /threads/time/average

Counter type	/threads/time/average
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average time spent executing one *HPX*-thread should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the average time spent executing one *HPX*-thread should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the average time spent executing one *HPX*-thread on the given *locality* since application start. If the instance name is total the counter returns the average time spent executing one *HPX*-thread for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the average time spent executing one *HPX*-thread for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_THREAD_CUMULATIVE_COUNTS (default: ON) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].

Table 2.53: Thread manager performance counter /threads/time/average-overhead

Counter type	/threads/time/average-overhead
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average overhead spent executing one *HPX*-thread should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the average overhead spent executing one *HPX*-thread should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the average time spent on overhead while executing one *HPX*-thread on the given *locality* since application start. If the instance name is total the counter returns the average time spent on overhead while executing one *HPX*-thread for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the average time spent on overhead executing one *HPX*-thread for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_THREAD_CUMULATIVE_COUNTS (default: ON) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].

Table 2.54: Thread manager performance counter /threads/count/cumulative-phases

Counter type	/threads/count/cumulative-phases
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the overall number of executed *HPX*-thread phases (invocations) should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the overall number of executed *HPX*-thread phases (invocations) should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the overall number of executed *HPX*-thread phases (invocations) on the given *locality* since application start. If the instance name is total the counter returns the accumulated number of executed *HPX*-thread phases (invocations) for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the overall number of executed *HPX*-thread phases for all worker threads separately. This counter is available only if the configuration time constant HPX_WITH_THREAD_CUMULATIVE_COUNTS is set to ON (default: ON). The unit of measure for this counter is nanosecond [ns].

Table 2.55: Thread manager performance counter /threads/time/average-phase

Counter type	/threads/time/average-phase
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average time spent executing one *HPX*-thread phase (invocation) should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the average time executing one *HPX*-thread phase (invocation) should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the average time spent executing one *HPX*-thread phase (invocation) on the given *locality* since application start. If the instance name is total the counter returns the average time spent executing one *HPX*-thread phase (invocation) for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the average time spent executing one *HPX*-thread phase for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_THREAD_CUMULATIVE_COUNTS (default: ON) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].

Table 2.56: Thread manager performance counter `/threads/time/average-phase-overhead`

Counter type	`/threads/time/average-phase-overhead`
Counter instance formatting	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the average time overhead executing one *HPX*-thread phase (invocation) should be queried for. The *locality* id (given by the `*`) is a (zero based) number identifying the *locality*. `pool#*` is defining the pool for which the current value of the idle-loop counter should be queried for. `worker-thread#*` is defining the worker thread for which the average overhead executing one *HPX*-thread phase (invocation) should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the average time spent on overhead executing one *HPX*-thread phase (invocation) on the given *locality* since application start. If the instance name is `total` the counter returns the average time spent on overhead while executing one *HPX*-thread phase (invocation) for all worker threads (cores) on that *locality*. If the instance name is `worker-thread#*` the counter will return the average time spent on overhead executing one *HPX*-thread phase for all worker threads separately. This counter is available only if the configuration time constants `HPX_WITH_THREAD_CUMULATIVE_COUNTS` (default: `ON`) and `HPX_WITH_THREAD_IDLE_RATES` are set to `ON` (default: `OFF`). The unit of measure for this counter is nanosecond [ns].

Table 2.57: Thread manager performance counter `/threads/time/overall`

Counter type	`/threads/time/overall`
Counter instance formatting	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the overall time spent running the scheduler should be queried for. The *locality* id (given by the `*`) is a (zero based) number identifying the *locality*. `pool#*` is defining the pool for which the current value of the idle-loop counter should be queried for. `worker-thread#*` is defining the worker thread for which the overall time spent running the scheduler should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the overall time spent running the scheduler on the given *locality* since application start. If the instance name is `total` the counter returns the overall time spent running the scheduler for all worker threads (cores) on that *locality*. If the instance name is `worker-thread#*` the counter will return the overall time spent running the scheduler for all worker threads separately. This counter is available only if the configuration time constant `HPX_WITH_THREAD_IDLE_RATES` is set to `ON` (default: `OFF`). The unit of measure for this counter is nanosecond [ns].

Table 2.58: Thread manager performance counter /threads/time/cumulative

Counter type	/threads/time/cumulative
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the overall time spent executing all *HPX*-threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the overall time spent executing all *HPX*-threads should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the overall time spent executing all *HPX*-threads on the given *locality* since application start. If the instance name is total the counter returns the overall time spent executing all *HPX*-threads for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the overall time spent executing all *HPX*-threads for all worker threads separately. This counter is available only if the configuration time constants HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS (default: ON) and HPX_THREAD_MAINTAIN_IDLE_RATES are set to ON (default: OFF).

Table 2.59: Thread manager performance counter /threads/time/cumulative-overheads

Counter type	/threads/time/cumulative-overheads
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the overall overhead time incurred by executing all *HPX*-threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the the overall overhead time incurred by executing all *HPX*-threads should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the overall overhead time incurred executing all *HPX*-threads on the given *locality* since application start. If the instance name is total the counter returns the overall overhead time incurred executing all *HPX*-threads for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the overall overhead time incurred executing all *HPX*-threads for all worker threads separately. This counter is available only if the configuration time constants HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS (default: ON) and HPX_THREAD_MAINTAIN_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].

Table 2.60: Thread manager performance counter threads/count/instantaneous/<thread-state>

Counter type	threads/count/instantaneous/<thread-state> where: <thread-state> is one of the following: all, active, pending, suspended, terminated, staged
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the current number of threads with the given state should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current number of threads with the given state should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool. The staged thread state refers to registered tasks before they are converted to thread objects.
Description	Returns the current number of *HPX*-threads having the given thread state on the given *locality*. If the instance name is total the counter returns the current number of *HPX*-threads of the given state for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the current number of *HPX*-threads in the given state for all worker threads separately.

Table 2.61: Thread manager performance counter threads/wait-time/<thread-state>

Counter type	threads/wait-time/<thread-state> where: <thread-state> is one of the following: pending staged
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average wait time of *HPX*-threads (pending) or thread descriptions (staged) with the given state should be queried for. The *locality* id (given by * is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the average wait time for the given state should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool. The staged thread state refers to the wait time of registered tasks before they are converted into thread objects, while the pending thread state refers to the wait time of threads in any of the scheduling queues.
Description	Returns the average wait time of *HPX*-threads (if the thread state is pending or of task descriptions (if the thread state is staged on the given *locality* since application start. If the instance name is total the counter returns the wait time of *HPX*-threads of the given state for all worker threads (cores) on that *locality*. If the instance name is worker-thread#* the counter will return the wait time of *HPX*-threads in the given state for all worker threads separately. These counters are available only if the compile time constant HPX_WITH_THREAD_QUEUE_WAITTIME was defined while compiling the *HPX* core library (default: OFF). The unit of measure for this counter is nanosecond [ns].

Table 2.62: Thread manager performance counter /threads/idle-rate

Counter type	/threads/idle-rate
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average idle rate of all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality* pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the averaged idle rate should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the average idle rate for the given worker thread(s) on the given *locality*. The idle rate is defined as the ratio of the time spent on scheduling and management tasks and the overall time spent executing work since the application started. This counter is available only if the configuration time constant HPX_WITH_THREAD_IDLE_RATES is set to ON (default: OFF).

Table 2.63: Thread manager performance counter /threads/creation-idle-rate

Counter type	/threads/creation-idle-rate
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average creation idle rate of all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the averaged idle rate should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the average idle rate for the given worker thread(s) on the given *locality* which is caused by creating new threads. The creation idle rate is defined as the ratio of the time spent on creating new threads and the overall time spent executing work since the application started. This counter is available only if the configuration time constants HPX_WITH_THREAD_IDLE_RATES (default: OFF) and HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES are set to ON.

Table 2.64: Thread manager performance counter /threads/cleanup-idle-rate

Counter type	/threads/cleanup-idle-rate
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average cleanup idle rate of all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the averaged cleanup idle rate should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the average idle rate for the given worker thread(s) on the given *locality* which is caused by cleaning up terminated threads. The cleanup idle rate is defined as the ratio of the time spent on cleaning up terminated thread objects and the overall time spent executing work since the application started. This counter is available only if the configuration time constants HPX_WITH_THREAD_IDLE_RATES (default: OFF) and HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES are set to ON.

Table 2.65: Thread manager performance counter /threadqueue/length

Counter type	/threadqueue/length
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the current length of all thread queues in the scheduler for all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current length of all thread queues in the scheduler should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the overall length of all queues for the given worker thread(s) on the given *locality*.

Table 2.66: Thread manager performance counter /threads/count/stack-unbinds

Counter type	/threads/count/stack-unbinds
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the unbind (madvise) operations should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the total number of *HPX*-thread unbind (madvise) operations performed for the referenced *locality*. Note that this counter is not available on Windows based platforms.

Table 2.67: Thread manager performance counter /threads/count/stack-recycles

Counter type	/threads/count/stack-recycles
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the recycling operations should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the total number of *HPX*-thread recycling operations performed.

Table 2.68: Thread manager performance counter /threads/count/stolen-from-pending

Counter type	/threads/count/stolen-from-pending
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the number of 'stole' threads should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the total number of *HPX*-threads 'stolen' from the pending thread queue by a neighboring thread worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).

Table 2.69: Thread manager performance counter /threads/count/pending-misses

Counter type	/threads/count/pending-misses
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the number of pending queue misses of all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality* pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the number of pending queue misses should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the total number of times that the referenced worker-thread on the referenced *locality* failed to find pending *HPX*-threads in its associated queue. This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).

Table 2.70: Thread manager performance counter /threads/count/pending-accesses

Counter type	/threads/count/pending-accesses
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the number of pending queue accesses of all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality* pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the number of pending queue accesses should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the total number of times that the referenced worker-thread on the referenced *locality* looked for pending *HPX*-threads in its associated queue. This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).

Table 2.71: Thread manager performance counter /threads/count/stolen-from-staged

Counter type	/threads/count/stolen-from-staged
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the number of *HPX*-threads stolen from the staged queue of all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the number of *HPX*-threads stolen from the staged queue should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the total number of *HPX*-threads 'stolen' from the staged thread queue by a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).

Table 2.72: Thread manager performance counter /threads/count/stolen-to-pending

Counter type	/threads/count/stolen-to-pending
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the number of *HPX*-threads stolen to the pending queue of all (or one) worker threads should be queried for. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the number of *HPX*-threads stolen to the pending queue should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option --hpx:threads. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the total number of *HPX*-threads 'stolen' to the pending thread queue of the worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).

Table 2.73: Thread manager performance counter /threads/count/stolen-to-staged

Counter type	/threads/count/stolen-to-staged
Counter instance formatting	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the number of *HPX*-threads stolen to the staged queue of all (or one) worker threads should be queried for. The *locality* id (given by *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the number of *HPX*-threads stolen to the staged queue should be queried for. The worker thread number (given by the *) is a (zero based) worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the total number of *HPX*-threads 'stolen' to the staged thread queue of a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).

Table 2.74: Thread manager performance counter /threads/count/objects

Counter type	/threads/count/objects
Counter instance formatting	locality#*/total or locality#*/allocator#* where: locality#* is defining the *locality* for which the current (cumulative) number of all created *HPX*-thread objects should be queried for. The *locality* id (given by *) is a (zero based) number identifying the *locality*. allocator#* is defining the number of the allocator instance using which the threads have been created. *HPX* uses a varying number of allocators to create (and recycle) *HPX*-thread objects, most likely these counters are of use for debugging purposes only. The allocator id (given by *) is a (zero based) number identifying the allocator to query.
Description	Returns the total number of *HPX*-thread objects created. Note that thread objects are reused to improve system performance, thus this number does not reflect the number of actually executed (retired) *HPX*-threads.

Table 2.75: Thread manager performance counter /scheduler/utilization/instantaneous

Counter type	/scheduler/utilization/instantaneous
Counter instance formatting	locality#*/total where: locality#* is defining the *locality* for which the current (instantaneous) scheduler utilization queried for. The *locality* id (given by *) is a (zero based) number identifying the *locality*.
Description	**Returns the total (instantaneous) scheduler utilization. This is the** current percentage of scheduler threads executing *HPX* threads.
Parameters	Percent

Table 2.76: Thread manager performance counter /threads/idle-loop-count/instantaneous

Counter type	/threads/idle-loop-count/instantaneous
Counter instance formatting	locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the current current accumulated value of all idle-loop counters of all worker threads should be queried. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current value of the idle-loop counter should be queried for. The worker thread number (given by the *) is a (zero based) worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option `--hpx:threads`. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the current (instantaneous) idle-loop count for the given *HPX*- worker thread or the accumulated value for all worker threads.

Table 2.77: Thread manager performance counter /threads/busy-loop-count/instantaneous

Counter type	/threads/busy-loop-count/instantaneous
Counter instance formatting	locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the current current accumulated value of all busy-loop counters of all worker threads should be queried. The *locality* id (given by the *) is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current value of the busy-loop counter should be queried for. The worker thread number (given by the *) is a (zero based) worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.
Description	Returns the current (instantaneous) busy-loop count for the given *HPX*- worker thread or the accumulated value for all worker threads.

Table 2.78: Thread manager performance counter /threads/time/background-work-duration

Counter type	/threads/time/background-work-duration
Counter instance formatting	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.
Description	Returns the overall time spent performing background work on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].

Table 2.79: Thread manager performance counter /threads/background-overhead

Counter type	/threads/background-overhead
Counter instance formatting	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the background overhead should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the background overhead should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.
Description	Returns the background overhead on the given locality since application start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%.

Table 2.80: Thread manager performance counter /threads/time/background-send-duration

Counter type	/threads/time/background-send-duration
Counter instance formatting	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work related to sending parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work related to sending parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.
Description	Returns the overall time spent performing background work related to sending parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcelport only.

Table 2.81: Thread manager performance counter /threads/background-send-overhead

Counter type	/threads/background-send-overhead
Counter instance formatting	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the background overhead related to sending parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the background overhead related to sending parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.
Description	Returns the background overhead related to sending parcels on the given locality since application start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcelport only.

Table 2.82: Thread manager performance counter /threads/time/background-receive-duration

Counter type	/threads/time/background-receive-duration
Counter instance formatting	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work related to receiving parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work related to receiving parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.
Description	Returns the overall time spent performing background work related to receiving parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcelport only.

Table 2.83: Thread manager performance counter `/threads/background-receive-overhead`

Counter type	`/threads/background-receive-overhead`
Counter instance formatting	`locality#*/total` or `locality#*/worker-thread#*` where: `locality#*` is defining the locality for which the background overhead related to receiving should be queried for. The locality id (given by `*`) is a (zero based) number identifying the locality. `worker-thread#*` is defining the worker thread for which the background overhead related to receiving parcels should be queried for. The worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *`--hpx:threads`*.
Description	Returns the background overhead related to receiving parcels on the given locality since application start. If the instance name is `total` the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is `worker-thread#*` the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants `HPX_WITH_BACKGROUND_THREAD_COUNTERS` (default: `OFF`) and `HPX_WITH_THREAD_IDLE_RATES` are set to `ON` (default: `OFF`). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcelport only.

Table 2.84: General performance counter `/runtime/count/component`

Counter type	`/runtime/count/component`
Counter instance formatting	`locality#*/total` where: `*` is the *locality* id of the *locality* the number of components should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall number of currently active components of the specified type on the given *locality*.
Parameters	The type of the component. This is the string which has been used while registering the component with *HPX*, e.g. which has been passed as the second parameter to the macro *`HPX_REGISTER_COMPONENT`*.

Table 2.85: General performance counter `/runtime/count/action-invocation`

Counter type	`/runtime/count/action-invocation`
Counter instance formatting	`locality#*/total` where: `*` is the *locality* id of the locality the number of action invocations should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall (local) invocation count of the specified action type on the given *locality*.
Parameters	The action type. This is the string which has been used while registering the action with *HPX*, e.g. which has been passed as the second parameter to the macro *`HPX_REGISTER_ACTION`* or *`HPX_REGISTER_ACTION_ID`*.

Table 2.86: General performance counter /runtime/count/remote-action-invocation

Counter type	/runtime/count/remote-action-invocation
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the number of action invocations should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall (remote) invocation count of the specified action type on the given *locality*.
Parameters	The action type. This is the string which has been used while registering the action with *HPX*, e.g. which has been passed as the second parameter to the macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID*.

Table 2.87: General performance counter /runtime/uptime

Counter type	/runtime/uptime
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the system uptime should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the overall time since application start on the given *locality* in nanoseconds.

Table 2.88: General performance counter /runtime/memory/virtual

Counter type	/runtime/memory/virtual
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the allocated virtual memory should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the amount of virtual memory currently allocated by the referenced *locality* (in bytes).

Table 2.89: General performance counter /runtime/memory/resident

Counter type	/runtime/memory/resident
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the allocated resident memory should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the amount of resident memory currently allocated by the referenced *locality* (in bytes).

Table 2.90: General performance counter /runtime/memory/total

Counter type	/runtime/memory/total
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the total available memory should be queried. The *locality* id is a (zero based) number identifying the *locality*. Note: only supported in Linux.
Description	**Returns the total available memory for use by the referenced** *locality* (in bytes). This counter is available on Linux and Windows systems only.

Table 2.91: General performance counter /runtime/io/read_bytes_issued

Counter type	/runtime/io/read_bytes_issued
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the number of bytes read should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of bytes read by the process (aggregate of count arguments passed to read() call or its analogues). This performance counter is available only on systems which expose the related data through the /proc file system.

Table 2.92: General performance counter /runtime/io/write_bytes_issued

Counter type	/runtime/io/write_bytes_issued
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the number of bytes written should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of bytes written by the process (aggregate of count arguments passed to write() call or its analogues). This performance counter is available only on systems which expose the related data through the /proc file system.

Table 2.93: General performance counter /runtime/io/read_syscalls

Counter type	/runtime/io/read_syscalls
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the number of system calls should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of system calls that perform I/O reads. This performance counter is available only on systems which expose the related data through the /proc file system.

Table 2.94: General performance counter `/runtime/io/write_syscalls`

Counter type	`/runtime/io/write_syscalls`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the number of system calls should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of system calls that perform I/O writes. This performance counter is available only on systems which expose the related data through the /proc file system.

Table 2.95: General performance counter `/runtime/io/read_bytes_transferred`

Counter type	`/runtime/io/read_bytes_transferred`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the number of bytes transferred should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of bytes retrieved from storage by I/O operations. This performance counter is available only on systems which expose the related data through the /proc file system.

Table 2.96: General performance counter `/runtime/io/write_bytes_transferred`

Counter type	`/runtime/io/write_bytes_transferred`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the number of bytes transferred should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of bytes retrieved from storage by I/O operations. This performance counter is available only on systems which expose the related data through the /proc file system.

Table 2.97: General performance counter `/runtime/io/write_bytes_cancelled`

Counter type	`/runtime/io/write_bytes_cancelled`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the number of bytes not being transferred should be queried. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of bytes accounted by write_bytes_transferred that has not been ultimately stored due to truncation or deletion. This performance counter is available only on systems which expose the related data through the /proc file system.

Table 2.98: Performance counter /papi/<papi_event>

Counter type	/papi/<papi_event> where: <papi_event> is the name of the PAPI event to expose as a performance counter (such as PAPI_SR_INS). Note that the list of available PAPI events changes depending on the used architecture. For a full list of available PAPI events and their (short) description use the --hpx:list-counters and --hpx:papi-event-info=all command line options.
Counter instance formatting	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the *locality* for which the current current accumulated value of all busy-loop counters of all worker threads should be queried. The *locality* id (given by *) is a (zero based) number identifying the *locality*. worker-thread#* is defining the worker thread for which the current value of the busy-loop counter should be queried for. The worker thread number (given by the *) is a (zero based) worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.
Description	Returns the current count of occurrences of the specified PAPI event. This counter is available only if the configuration time constant HPX_WITH_PAPI is set to ON (default: OFF).

Table 2.99: Performance counter /statistics/average

Counter type	/statistics/average
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume 1000 [ms] as the default. The second value can be either 0 or 1 and specifies whether the underlying counter should be reset during evaluation 1 or not 0. The default value is 0.

Table 2.100: Performance counter /statistics/rolling_average

Counter type	/statistics/rolling_average
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current rolling average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to three comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume 1000 [ms] as the default. The second value will be interpreted as the size of the rolling window (the number of latest values to use to calculate the rolling average). The default value for this is 10. The third value can be either 0 or 1 and specifies whether the underlying counter should be reset during evaluation 1 or not 0. The default value is 0.

Table 2.101: Performance counter `/statistics/stddev`

Counter type	`/statistics/stddev`
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current standard deviation (stddev) value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.102: Performance counter `/statistics/rolling_stddev`

Counter type	`/statistics/rolling_stddev`
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current rolling variance (stddev) value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to three comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value will be interpreted as the size of the rolling window (the number of latest values to use to calculate the rolling average). The default value for this is `10`. The third value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.103: Performance counter `/statistics/median`

Counter type	`/statistics/median`
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current (statistically estimated) median value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.104: Performance counter `/statistics/max`

Counter type	`/statistics/max`
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current maximum value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.105: Performance counter `/statistics/rolling_max`

Counter type	`/statistics/rolling_max`
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current rolling maximum value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to three comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value will be interpreted as the size of the rolling window (the number of latest values to use to calculate the rolling average). The default value for this is `10`. The third value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.106: Performance counter `/statistics/min`

Counter type	`/statistics/min`
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current minimum value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.107: Performance counter `/statistics/rolling_min`

Counter type	`/statistics/rolling_min`
Counter instance formatting	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.
Description	Returns the current rolling minimum value calculated based on the values queried from the underlying counter (the one specified as the instance name).
Parameters	Any parameter will be interpreted as a list of up to three comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value will be interpreted as the size of the rolling window (the number of latest values to use to calculate the rolling average). The default value for this is `10`. The third value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.108: Performance counter `/arithmetics/add`

Counter type	`/arithmetics/add`
Description	Returns the sum calculated based on the values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.109: Performance counter `/arithmetics/subtract`

Counter type	`/arithmetics/subtract`
Description	Returns the difference calculated based on the values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.110: Performance counter `/arithmetics/multiply`

Counter type	`/arithmetics/multiply`
Description	Returns the product calculated based on the values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.111: Performance counter `/arithmetics/divide`

Counter type	`/arithmetics/divide`
Description	Returns the result of division of the values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.112: Performance counter `/arithmetics/mean`

Counter type	`/arithmetics/mean`
Description	Returns the average value of all values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.113: Performance counter `/arithmetics/variance`

Counter type	`/arithmetics/variance`
Description	Returns the standard deviation of all values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.114: Performance counter `/arithmetics/median`

Counter type	`/arithmetics/median`
Description	Returns the median value of all values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.115: Performance counter `/arithmetics/min`

Counter type	`/arithmetics/min`
Description	Returns the minimum value of all values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.116: Performance counter `/arithmetics/max`

Counter type	`/arithmetics/max`
Description	Returns the maximum value of all values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Table 2.117: Performance counter `/arithmetics/count`

Counter type	`/arithmetics/count`
Description	Returns the count value of all values queried from the underlying counters (the ones specified as the parameters).
Parameters	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Note: The `/arithmetics` counters can consume an arbitrary number of other counters. For this reason those have to be specified as parameters (a comma separated list of counters appended after a `'@'`). For instance:

```
$ ./bin/hello_world_distributed -t2 \
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
    --hpx:print-counter=/arithmetics/add@/threads{locality#0/worker-thread#*}/count/
↪cumulative
hello world from OS-thread 0 on locality 0
hello world from OS-thread 1 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.515640,[s],25
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.515520,[s],36
/arithmetics/add@/threads{locality#0/worker-thread#*}/count/cumulative,1,0.516445,[s],64
```

Since all wildcards in the parameters are expanded, this example is fully equivalent to specifying both counters separately to `/arithmetics/add`:

```
$ ./bin/hello_world_distributed -t2 \
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
    --hpx:print-counter=/arithmetics/add@\
        /threads{locality#0/worker-thread#0}/count/cumulative,\
        /threads{locality#0/worker-thread#1}/count/cumulative
```

Table 2.118: Performance counter `/coalescing/count/parcels`

Counter type	`/coalescing/count/parcels`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the number of parcels for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of parcels handled by the message handler associated with the action which is given by the counter parameter.
Parameters	The action type. This is the string which has been used while registering the action with *HPX*, e.g. which has been passed as the second parameter to the macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID*.

Table 2.119: Performance counter `/coalescing/count/messages`

Counter type	`/coalescing/count/messages`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the number of messages for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the number of messages generated by the message handler associated with the action which is given by the counter parameter.
Parameters	The action type. This is the string which has been used while registering the action with *HPX*, e.g. which has been passed as the second parameter to the macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID*.

Table 2.120: Performance counter `/coalescing/count/average-parcels-per-message`

Counter type	`/coalescing/count/average-parcels-per-message`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the number of messages for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the average number of parcels sent in a message generated by the message handler associated with the action which is given by the counter parameter.
Parameters	The action type. This is the string which has been used while registering the action with *HPX*, e.g. which has been passed as the second parameter to the macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID*

Table 2.121: Performance counter `/coalescing/time/average-parcel-arrival`

Counter type	`/coalescing/time/average-parcel-arrival`
Counter instance formatting	`locality#*/total` where: * is the *locality* id of the *locality* the average time between parcels for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns the average time between arriving parcels for the action which is given by the counter parameter.
Parameters	The action type. This is the string which has been used while registering the action with *HPX*, e.g. which has been passed as the second parameter to the macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID*

Table 2.122: Performance counter /coalescing/time/parcel-arrival-histogram

Counter type	/coalescing/time/parcel-arrival-histogram
Counter instance formatting	locality#*/total where: * is the *locality* id of the *locality* the average time between parcels for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.
Description	Returns a histogram representing the times between arriving parcels for the action which is given by the counter parameter. This counter returns an array of values, where the first three values represent the three parameters used for the histogram followed by one value for each of the histogram buckets. The first unit of measure displayed for this counter [ns] refers to the lower and upper boundary values in the returned histogram data only. The second unit of measure displayed [0.1%] refers to the actual histogram data. For each bucket the counter shows a value between 0 and 1000 which corresponds to a percentage value between 0% and 100%.
Parameters	The action type and optional histogram parameters. The action type is the string which has been used while registering the action with *HPX*, e.g. which has been passed as the second parameter to the macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID*. The action type may be followed by a comma separated list of up-to three numbers: the lower and upper boundaries for the collected histogram, and the number of buckets for the histogram to generate. By default these three numbers will be assumed to be 0 ([ns], lower bound), 1000000 ([ns], upper bound), and 20 (number of buckets to generate).

Note: The performance counters related to *parcel* coalescing are available only if the configuration time constant HPX_WITH_PARCEL_COALESCING is set to ON (default: ON). However, even in this case it will be available only for actions that are enabled for parcel coalescing (see the macros HPX_ACTION_USES_MESSAGE_COALESCING and HPX_ACTION_USES_MESSAGE_COALESCING_NOTHROW).

APEX integration

HPX provides integration with APEX[166], which is a framework for application profiling using task timers and various performance counters Huck *et al.*[174]. It can be added as a git submodule by turning on the option *HPX_WITH_APEX:BOOL* during CMake[167] configuration. TAU[168] is an optional dependency when using APEX.

To build *HPX* with APEX[169], add *HPX_WITH_APEX*=ON, and, optionally, Tau_ROOT=$PATH_TO_TAU to your CMake[170] configuration. In addition, you can override the tag used for APEX[171] with the HPX_WITH_APEX_TAG option. Please see the APEX *HPX* documentation[172] for detailed instructions on using APEX[173] with *HPX*.

[166] http://uo-oaciss.github.io/apex
[174] K. A. Huck, A. Porterfield, N Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler. *An autonomic performance environment for exascale*. Supercomputing Frontiers and Innovations, 2015.
[167] https://www.cmake.org
[168] https://www.cs.uoregon.edu/research/tau/home.php
[169] http://uo-oaciss.github.io/apex
[170] https://www.cmake.org
[171] http://uo-oaciss.github.io/apex
[172] https://uo-oaciss.github.io/apex/usage/#hpx-louisiana-state-university
[173] http://uo-oaciss.github.io/apex

References

2.3.15 Using the LCI parcelport

Basic information

The Lightweight Communication Interface[175] (LCI) is an ongoing research project aiming to provide efficient support for applications with irregular and asynchronous communication patterns such as graph analysis, sparse linear algebra, and task-based runtime on modern parallel architectures. Its features include (a) support for more communication primitives such as two-sided send/recv and one-sided (dynamic or direct) remote put/get (b) better multi-threaded performance (c) explicit user control of communication resource (d) flexible signaling mechanisms such as synchronizer, completion queue, and active message handler. It is designed to be a low-level communication library used by high-level libraries and frameworks.

The LCI parcelport is an experimental parcelport. It aims to provide the best possible communication performance on high-performance computation platforms. Compared to the MPI parcelport, it uses much fewer messages and memory copies to transfer an *HPX* parcel over the network. Its message transmission path involves minimum synchronization points and is almost lock-free. It is expected to be much faster than the MPI parcelport.

Build *HPX* with the LCI parcelport

While building *HPX*, you can specify a set of CMake[176] variables to enable and configure the LCI parcelport. Below, there is a set of the most important and frequently used CMake variables.

`HPX_WITH_PARCELPORT_LCI`

Enable the LCI parcelport. This enables the use of LCI for networking operations in the *HPX* runtime. The default value is `OFF` because it's not available on all systems and/or requires another dependency. However, this experimental parcelport may provide better performance than the MPI parcelport. You must set this variable to `ON` in order to use the LCI parcelport. All the following variables only make sense when this variable is set to `ON`.

`HPX_WITH_FETCH_LCI`

Use FetchContent to fetch LCI. The default value is `OFF`. If this option is set to `OFF`. You need to install your own LCI library and *HPX* will try to find it using CMake[177] `find_package`. You can specify the location of the LCI installation by the environmental variable `LCI_ROOT`. Refer to the LCI README[178] for how to install LCI. If this option is set to `ON`. *HPX* will fetch and build LCI for you. You can use the following CMake[179] variables to configure this behavior for your platform.

`HPX_WITH_LCI_TAG`

This variable only takes effect when `HPX_WITH_FETCH_LCI` is set to `ON` and `FETCHCONTENT_SOURCE_DIR_LCI` is not set. *HPX* will fetch LCI from its github repository. This variable controls the branch/tag LCI will be fetched.

`FETCHCONTENT_SOURCE_DIR_LCI`

This variable only takes effect when `HPX_WITH_FETCH_LCI` is set to `ON`. When it is defined, `HPX_WITH_LCI_TAG` will be ignored. It accepts a path to a local version of LCI source code and *HPX* will fetch and build LCI from there. The default value is set conservatively for the stability of *HPX*, but users are welcome to set this variable to `master` for potentially better performance.

175 https://github.com/uiuc-hpc/lci
176 https://www.cmake.org
177 https://www.cmake.org
178 https://github.com/uiuc-hpc/lci#readme
179 https://www.cmake.org

Run *HPX* with the LCI parcelport

We use the same mechanisms as MPI to launch LCI, so you can use the same way you run MPI parcelport to run LCI parcelport. Typically, it would be `hpxrun`, `mpirun`, or `srun`.

If you are using `hpxrun.py`, just pass `--parcelport lci` to the scripts.

If you are using `mpirun` or `srun`, you can just pass `--hpx:ini=hpx.parcel.lci.priority=1000`, `--hpx:ini=hpx.parcel.lci.enable=1`, and `--hpx:ini=hpx.parcel.bootstrap=lci` to the *HPX* applications.

Performance tuning of the LCI parcelport

We encourage users to set the following environmental variables when using the LCI parcelport to get better performance.

```
$ export LCI_SERVER_MAX_SENDS=1024
$ export LCI_SERVER_MAX_RECVS=4096
$ export LCI_SERVER_NUM_PKTS=65536
$ export LCI_SERVER_MAX_CQES=65536
$ export LCI_PACKET_SIZE=12288
```

This setting needs roughly 800MB memory per process. The memory consumption mainly comes from the packets, which can be calculated using *LCI_SERVER_NUM_PKTS x LCI_PACKET_SIZE*.

2.3.16 *HPX* runtime and resources

HPX thread scheduling policies

The *HPX* runtime has six thread scheduling policies: local-priority, static-priority, local, static, local-workrequesting-fifo, and abp-priority. These policies can be specified from the command line using the command line option *`--hpx:queuing`*. In order to use a particular scheduling policy, the runtime system must be built with the appropriate scheduler flag turned on (e.g. `cmake -DHPX_THREAD_SCHEDULERS=local`, see *CMake options* for more information).

Priority local scheduling policy (default policy)

The priority local scheduling policy maintains one queue per operating system (OS) thread. The OS thread pulls its work from this queue. By default the number of high priority queues is equal to the number of OS threads; the number of high priority queues can be specified on the command line using *`--hpx:high-priority-threads`*. High priority threads are executed by any of the OS threads before any other work is executed. When a queue is empty, work will be taken from high priority queues first. There is one low priority queue from which threads will be scheduled only when there is no other work.

For this scheduling policy there is an option to turn on NUMA sensitivity using the command line option *`--hpx:numa-sensitive`*. When NUMA sensitivity is turned on, work stealing is done from queues associated with the same NUMA domain first, only after that work is stolen from other NUMA domains.

This scheduler is enabled at build time by default using the FIFO (first-in-first-out) queueing policy. This policy can be invoked using *`--hpx:queuing`*`local-priority-fifo`. The scheduler can also be enabled using the LIFO (last-in-first-out) policy. This is not the default policy and must be invoked using the command line option *`--hpx:queuing`*`local-priority-lifo`.

Static priority scheduling policy

- invoke using: `--hpx:queuing` `static-priority` (or `-qs`)

The static scheduling policy maintains one queue per OS thread from which each OS thread pulls its tasks (user threads). Threads are distributed in a round robin fashion. There is no thread stealing in this policy.

Local scheduling policy

- invoke using: `--hpx:queuing` `local` (or `-ql`)
- flag to turn on for build: `HPX_THREAD_SCHEDULERS=all` or `HPX_THREAD_SCHEDULERS=local`

The local scheduling policy maintains one queue per OS thread from which each OS thread pulls its tasks (user threads).

Static scheduling policy

- invoke using: `--hpx:queuing` `static`
- flag to turn on for build: `HPX_THREAD_SCHEDULERS=all` or `HPX_THREAD_SCHEDULERS=static`

The static scheduling policy maintains one queue per OS thread from which each OS thread pulls its tasks (user threads). Threads are distributed in a round robin fashion. There is no thread stealing in this policy.

Priority ABP scheduling policy

- invoke using: `--hpx:queuing` `abp-priority-fifo`
- flag to turn on for build: `HPX_THREAD_SCHEDULERS=all` or `HPX_THREAD_SCHEDULERS=abp-priority`

Priority ABP policy maintains a double ended lock free queue for each OS thread. By default the number of high priority queues is equal to the number of OS threads; the number of high priority queues can be specified on the command line using `--hpx:high-priority-threads`. High priority threads are executed by the first OS threads before any other work is executed. When a queue is empty work will be taken from high priority queues first. There is one low priority queue from which threads will be scheduled only when there is no other work. For this scheduling policy there is an option to turn on NUMA sensitivity using the command line option `--hpx:numa-sensitive`. When NUMA sensitivity is turned on work stealing is done from queues associated with the same NUMA domain first, only after that work is stolen from other NUMA domains.

This scheduler can be used with two underlying queuing policies (FIFO: first-in-first-out, and LIFO: last-in-first-out). In order to use the LIFO policy use the command line option `--hpx:queuing` `abp-priority-lifo`.

Work requesting scheduling policies

- invoke using: `--hpx:queuing` `local-workrequesting-fifo`, using `--hpx:queuing` `local-workrequesting-lifo`, or using `--hpx:queuing` `local-workrequesting-mc`

The work-requesting policies rely on a different mechanism of balancing work between cores (compared to the other policies listed above). Instead of actively trying to steal work from other cores, requesting work relies on a less disruptive mechanism. If a core runs out of work, instead of actively looking at the queues of neighboring cores, in this case a request is posted to another core. This core now (whenever it is not busy with other work) either responds to the original core by sending back work or passes the request on to the next possible core in the system. In general, this scheme avoids contention on the work queues as those are always accessed by their own cores only.

The *HPX* resource partitioner

The *HPX* resource partitioner lets you take the execution resources available on a system—processing units, cores, and numa domains—and assign them to thread pools. By default *HPX* creates a single thread pool name `default`. While this is good for most use cases, the resource partitioner lets you create multiple thread pools with custom resources and options.

Creating custom thread pools is useful for cases where you have tasks which absolutely need to run without interference from other tasks. An example of this is when using MPI[180] for distribution instead of the built-in mechanisms in *HPX* (useful in legacy applications). In this case one can create a thread pool containing a single thread for MPI communication. MPI tasks will then always run on the same thread, instead of potentially being stuck in a queue behind other threads.

Note that *HPX* thread pools are completely independent from each other in the sense that task stealing will never happen between different thread pools. However, tasks running on a particular thread pool can schedule tasks on another thread pool.

Note: It is simpler in some situations to schedule important tasks with high priority instead of using a separate thread pool.

Using the resource partitioner

The `hpx::resource::partitioner` is now created during *HPX* runtime initialization without explicit action needed from the user. To specify some of the initialization parameters you can use the `hpx::init_params`.

The resource partitioner callback is the interface to add thread pools to the *HPX* runtime and to assign resources to the thread pools. In order to create custom thread pools you can specify the resource partitioner callback `hpx::init_params::rp_callback` which will be called once the resource partitioner will be created , see the example below. You can also specify other parameters, see `hpx::init_params`.

To add a thread pool use the `hpx::resource::partitioner::create_thread_pool` method. If you simply want to use the default scheduler and scheduler options, it is enough to call `rp.create_thread_pool("my-thread-pool")`.

Then, to add resources to the thread pool you can use the `hpx::resource::partitioner::add_resource` method. The resource partitioner exposes the hardware topology retrieved using Portable Hardware Locality (HWLOC)[181] and lets you iterate through the topology to add the wanted processing units to the thread pool. Below is an example of adding all processing units from the first NUMA domain to a custom thread pool, unless there is only one NUMA domain in which case we leave the first processing unit for the default thread pool:

Note: Whatever processing units are not assigned to a thread pool by the time `hpx::init` is called will be added to the default thread pool. It is also possible to explicitly add processing units to the default thread pool, and to create the default thread pool manually (in order to e.g. set the scheduler type).

Tip: The command line option `--hpx:print-bind` is useful for checking that the thread pools have been set up the way you expect.

[180] https://en.wikipedia.org/wiki/Message_Passing_Interface
[181] https://www.open-mpi.org/projects/hwloc/

Difference between the old and new version

In the old version, you had to create an instance of the `resource_partitioner` with `argc` and `argv`.

```
int main(int argc, char** argv)
{
    hpx::resource::partitioner rp(argc, argv);
    hpx::init();
}
```

From *HPX* 1.5.0 onwards, you just pass `argc` and `argv` to `hpx::init()` or `hpx::start()` for the binding options to be parsed by the resource partitioner.

```
int main(int argc, char** argv)
{
    hpx::init_params init_args;
    hpx::init(argc, argv, init_args);
}
```

In the old version, when creating a custom thread pool, you just called the utilities on the resource partitioner instantiated previously.

```
int main(int argc, char** argv)
{
    hpx::resource::partitioner rp(argc, argv);

    rp.create_thread_pool("my-thread-pool");

    bool one_numa_domain = rp.numa_domains().size() == 1;
    bool skipped_first_pu = false;

    hpx::resource::numa_domain const& d = rp.numa_domains()[0];

    for (const hpx::resource::core& c : d.cores())
    {
        for (const hpx::resource::pu& p : c.pus())
        {
            if (one_numa_domain && !skipped_first_pu)
            {
                skipped_first_pu = true;
                continue;
            }

            rp.add_resource(p, "my-thread-pool");
        }
    }

    hpx::init();
}
```

You now specify the resource partitioner callback which will tie the resources to the resource partitioner created during runtime initialization.

```
void init_resource_partitioner_handler(hpx::resource::partitioner& rp)
{
    rp.create_thread_pool("my-thread-pool");

    bool one_numa_domain = rp.numa_domains().size() == 1;
    bool skipped_first_pu = false;

    hpx::resource::numa_domain const& d = rp.numa_domains()[0];

    for (const hpx::resource::core& c : d.cores())
    {
        for (const hpx::resource::pu& p : c.pus())
        {
            if (one_numa_domain && !skipped_first_pu)
            {
                skipped_first_pu = true;
                continue;
            }

            rp.add_resource(p, "my-thread-pool");
        }
    }
}

int main(int argc, char* argv[])
{
    hpx::init_params init_args;
    init_args.rp_callback = &init_resource_partitioner_handler;

    hpx::init(argc, argv, init_args);
}
```

Advanced usage

It is possible to customize the built in schedulers by passing scheduler options to `hpx::resource::partitioner::create_thread_pool`. It is also possible to create and use custom schedulers.

Note: It is not recommended to create your own scheduler. The *HPX* developers use this to experiment with new scheduler designs before making them available to users via the standard mechanisms of choosing a scheduler (command line options). If you would like to experiment with a custom scheduler the resource partitioner example `shared_priority_queue_scheduler.cpp` contains a fully implemented scheduler with logging, etc. to make exploration easier.

To choose a scheduler and custom mode for a thread pool, pass additional options when creating the thread pool like this:

```
rp.create_thread_pool("my-thread-pool",
    hpx::resource::policies::local_priority_lifo,
    hpx::policies::scheduler_mode(
```

(continues on next page)

(continued from previous page)

```
        hpx::policies::scheduler_mode::default |
        hpx::policies::scheduler_mode::enable_elasticity));
```

The available schedulers are documented here: `hpx::resource::scheduling_policy`, and the available scheduler modes here: `hpx::threads::policies::scheduler_mode`. Also see the examples folder for examples of advanced resource partitioner usage: `simple_resource_partitioner.cpp` and `oversubscribing_resource_partitioner.cpp`.

2.3.17 Miscellaneous

Error handling

Like in any other asynchronous invocation scheme, it is important to be able to handle error conditions occurring while the asynchronous (and possibly remote) operation is executed. In *HPX* all error handling is based on standard C++ exception handling. Any exception thrown during the execution of an asynchronous operation will be transferred back to the original invocation *locality*, where it will be rethrown during synchronization with the calling thread.

The source code for this example can be found here: `error_handling.cpp`.

Working with exceptions

For the following description assume that the function `raise_exception()` is executed by invoking the plain action `raise_exception_type`.

```
#include <hpx/iostream.hpp>
#include <hpx/modules/runtime_local.hpp>

//[error_handling_raise_exception
void raise_exception()
```

The exception is thrown using the macro *`HPX_THROW_EXCEPTION`*. The type of the thrown exception is *`hpx::exception`*. This associates additional diagnostic information with the exception, such as file name and line number, *locality* id and thread id, and stack backtrace from the point where the exception was thrown.

Any exception thrown during the execution of an action is transferred back to the (asynchronous) invocation site. It will be rethrown in this context when the calling thread tries to wait for the result of the action by invoking either `future<>::get()` or the synchronous action invocation wrapper as shown here:

```
{
    {
        ///////////////////////////////////////////////////////////////////////
        // Error reporting using exceptions
        //[exception_diagnostic_information
        hpx::cout << "Error reporting using exceptions\n";
        try
        {
            // invoke raise_exception() which throws an exception
            raise_exception_action do_it;
            do_it(hpx::find_here());
        }
        catch (hpx::exception const& e)
```

(continues on next page)

(continued from previous page)

```
        {
            // Print just the essential error information.
            hpx::cout << "caught exception: " << e.what() << "\n\n";
```

Note: The exception is transferred back to the invocation site even if it is executed on a different *locality*.

Additionally, this example demonstrates how an exception thrown by an (possibly remote) action can be handled. It shows the use of *hpx::diagnostic_information*, which retrieves all available diagnostic information from the exception as a formatted string. This includes, for instance, the name of the source file and line number, the sequence number of the OS thread and the *HPX* thread id, the *locality* id and the stack backtrace of the point where the original exception was thrown.

Under certain circumstances it is desirable to output only some of the diagnostics, or to output those using different formatting. For this case, *HPX* exposes a set of lower-level functions as demonstrated in the following code snippet:

```
                      << hpx::diagnostic_information(e) << "\n";
        }
        hpx::cout << std::flush;
        //]

        // Detailed error reporting using exceptions
        //[exception_diagnostic_elements
        hpx::cout << "Detailed error reporting using exceptions\n";
        try
        {
            // Invoke raise_exception() which throws an exception.
            raise_exception_action do_it;
            do_it(hpx::find_here());
        }
        catch (hpx::exception const& e)
        {
            // Print the elements of the diagnostic information separately.
            hpx::cout << "{what}: " << hpx::get_error_what(e) << "\n";
            hpx::cout << "{locality-id}: " << hpx::get_error_locality_id(e)
                      << "\n";
            hpx::cout << "{hostname}: " << hpx::get_error_host_name(e) << "\n";
            hpx::cout << "{pid}: " << hpx::get_error_process_id(e) << "\n";
            hpx::cout << "{function}: " << hpx::get_error_function_name(e)
                      << "\n";
            hpx::cout << "{file}: " << hpx::get_error_file_name(e) << "\n";
            hpx::cout << "{line}: " << hpx::get_error_line_number(e) << "\n";
```

Working with error codes

Most of the API functions exposed by *HPX* can be invoked in two different modes. By default those will throw an exception on error as described above. However, sometimes it is desirable not to throw an exception in case of an error condition. In this case an object instance of the `hpx::error_code` type can be passed as the last argument to the API function. In case of an error, the error condition will be returned in that `hpx::error_code` instance. The following example demonstrates extracting the full diagnostic information without exception handling:

```
                  << "\n";
        hpx::cout << "{stack-trace}: " << hpx::get_error_backtrace(e)
                  << "\n";
        hpx::cout << "{env}: " << hpx::get_error_env(e) << "\n";
    }
    hpx::cout << std::flush;
    //]

    ///////////////////////////////////////////////////////////////////////
    // Error reporting using error code
    {
        //[error_handling_diagnostic_information
        hpx::cout << "Error reporting using error code\n";

        // Create a new error_code instance.
        hpx::error_code ec;

        // If an instance of an error_code is passed as the last argument while
        // invoking the action, the function will not throw in case of an error
        // but store the error information in this error_code instance instead.
        raise_exception_action do_it;
        do_it(hpx::find_here(), ec);
```

Note: The error information is transferred back to the invocation site even if it is executed on a different *locality*.

This example show how an error can be handled without having to resolve to exceptions and that the returned `hpx::error_code` instance can be used in a very similar way as the `hpx::exception` type above. Simply pass it to the `hpx::diagnostic_information`, which retrieves all available diagnostic information from the error code instance as a formatted string.

As for handling exceptions, when working with error codes, under certain circumstances it is desirable to output only some of the diagnostics, or to output those using different formatting. For this case, *HPX* exposes a set of lower-level functions usable with error codes as demonstrated in the following code snippet:

```
            // Print all of the available diagnostic information as stored with
            // the exception.
            hpx::cout << "diagnostic information:"
                      << hpx::diagnostic_information(ec) << "\n";
        }

        hpx::cout << std::flush;
        //]
    }
```

(continues on next page)

(continued from previous page)

```
        // Detailed error reporting using error code
        {
            //[error_handling_diagnostic_elements
            hpx::cout << "Detailed error reporting using error code\n";

            // Create a new error_code instance.
            hpx::error_code ec;

            // If an instance of an error_code is passed as the last argument while
            // invoking the action, the function will not throw in case of an error
            // but store the error information in this error_code instance instead.
            raise_exception_action do_it;
            do_it(hpx::find_here(), ec);

            if (ec)
            {
                // Print the elements of the diagnostic information separately.
                hpx::cout << "{what}: " << hpx::get_error_what(ec) << "\n";
                hpx::cout << "{locality-id}: " << hpx::get_error_locality_id(ec)
                          << "\n";
                hpx::cout << "{hostname}: " << hpx::get_error_host_name(ec)
                          << "\n";
                hpx::cout << "{pid}: " << hpx::get_error_process_id(ec) << "\n";
```

For more information please refer to the documentation of *hpx::get_error_what*, *hpx::get_error_locality_id*, *hpx::get_error_host_name*, *hpx::get_error_process_id*, *hpx::get_error_function_name*, *hpx::get_error_file_name*, *hpx::get_error_line_number*, *hpx::get_error_os_thread*, *hpx::get_error_thread_id*, *hpx::get_error_thread_description*, *hpx::get_error_backtrace*, *hpx::get_error_env*, and *hpx::get_error_state*.

Lightweight error codes

Sometimes it is not desirable to collect all the ambient information about the error at the point where it happened as this might impose too much overhead for simple scenarios. In this case, *HPX* provides a lightweight error code facility that will hold the error code only. The following snippet demonstrates its use:

```
                          << "\n";
                hpx::cout << "{thread-id}: " << std::hex
                          << hpx::get_error_thread_id(ec) << "\n";
                hpx::cout << "{thread-description}: "
                          << hpx::get_error_thread_description(ec) << "\n\n";
                hpx::cout << "{state}: " << std::hex << hpx::get_error_state(ec)
                          << "\n";
                hpx::cout << "{stack-trace}: " << hpx::get_error_backtrace(ec)
                          << "\n";
                hpx::cout << "{env}: " << hpx::get_error_env(ec) << "\n";
            }

            hpx::cout << std::flush;
            //]
        }
```

(continues on next page)

(continued from previous page)

```
        // Error reporting using lightweight error code
        {
            //[lightweight_error_handling_diagnostic_information
            hpx::cout << "Error reporting using an lightweight error code\n";
```

All functions that retrieve other diagnostic elements from the *hpx::error_code* will fail if called with a lightweight error_code instance.

Utilities in *HPX*

In order to ease the burden of programming, *HPX* provides several utilities to users. The following section documents those facilies.

Checkpoint

See *checkpoint*.

The *HPX* I/O-streams component

The *HPX* I/O-streams subsystem extends the standard C++ output streams `std::cout` and `std::cerr` to work in the distributed setting of an *HPX* application. All of the output streamed to `hpx::cout` will be dispatched to `std::cout` on the console *locality*. Likewise, all output generated from `hpx::cerr` will be dispatched to `std::cerr` on the console *locality*.

Note: All existing standard manipulators can be used in conjunction with `hpx::cout` and `hpx::cerr`.

In order to use either `hpx::cout` or `hpx::cerr`, application codes need to `#include <hpx/include/iostreams.hpp>`. For an example, please see the following 'Hello world' program:

```
//  Copyright (c) 2007-2012 Hartmut Kaiser
//
//  SPDX-License-Identifier: BSL-1.0
//  Distributed under the Boost Software License, Version 1.0. (See accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

///////////////////////////////////////////////////////////////////////////////
// The purpose of this example is to execute a HPX-thread printing
// "Hello World!" once. That's all.

//[hello_world_1_getting_started
// Including 'hpx/hpx_main.hpp' instead of the usual 'hpx/hpx_init.hpp' enables
// to use the plain C-main below as the direct main HPX entry point.
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

int main()
{
    // Say hello to the world!
```

(continues on next page)

(continued from previous page)

```
    hpx::cout << "Hello World!\n" << std::flush;
    return 0;
}
//]
```

Additionally, those applications need to link with the iostreams component. When using CMake this can be achieved by using the `COMPONENT_DEPENDENCIES` parameter; for instance:

```
include(HPX_AddExecutable)

add_hpx_executable(
    hello_world
    SOURCES hello_world.cpp
    COMPONENT_DEPENDENCIES iostreams
)
```

Note: The `hpx::cout` and `hpx::cerr` streams buffer all output locally until a `std::endl` or `std::flush` is encountered. That means that no output will appear on the console as long as either of these is explicitly used.

2.3.18 Troubleshooting

Common issues

This section contains commonly encountered problems when compiling or using HPX.

See also the closed issues on GitHub[182] to find out how other people resolved a similar problem. If nothing of that works, you can also open a new issue on GitHub[183] or contact us using one the options found in Support for deploying and using HPX[184].

`HPX::iostreams_component" target not found`

You may see a CMake[185] error message that looks a bit like this:

```
error: `HPX::iostreams_component`` target not found
```

Simply ensure that *HPX* is installed with `HPX_WITH_DISTRIBUTED_RUNTIME=ON` to prevent encountering such error(s). This is required if you want to use `hpx::cout`.

[182] https://github.com/STEllAR-GROUP/hpx/issues?q=is%3Aissue+is%3Aclosed
[183] https://github.com/STEllAR-GROUP/hpx/issues
[184] https://github.com/STEllAR-GROUP/hpx/blob/master/.github/SUPPORT.md
[185] https://www.cmake.org

Undefined reference to hpx::cout

You may see a linker error message that looks a bit like this:

```
hello_world.cpp:(.text+0x5aa): undefined reference to `hpx::cout'
```

This usually happens if you are trying to use *HPX* iostreams functionality such as `hpx::cout` but are not linking against it. The iostreams functionality is not part of the core *HPX* library, and must be linked to explicitly. Typically this can be solved by adding `COMPONENT_DEPENDENCIES iostreams` to a call to `add_hpx_library/add_hpx_executable/hpx_setup_target` if using CMake[186]. See *Creating HPX projects* for more details.

Fail compiling for examples with hpx::future and co_await

You may see an error message that looks a bit like this:

```
error: coroutines require a traits template; cannot find 'std::coroutine_traits'
```

This can be resolved by using `-DHPX_WITH_CXX_STANDARD=20` to the cmake command line. Note that a compiler that supports C++20 is needed.

See also the corresponding closed Issue #5784[187].

Build fails with ASIO error

You may see an error message that looks a bit like this:

```
Cannot open include file asio/io_context.hpp
```

This can be resolved by using `-DHPX_WITH_FETCH_ASIO=ON` to the cmake command line.

See also the corresponding closed Issue #5404[188] for more information.

Build fails with TCMalloc error

You may see an error message that looks a bit like this:

```
Could NOT find TCMalloc (missing: Tcmalloc_LIBRARY Tcmalloc_INCLUDE_DIR)
ERROR: HPX_WITH_MALLOC was set to tcmalloc, but tcmalloc could not be
found.  Valid options for HPX_WITH_MALLOC are: system, tcmalloc, jemalloc,
mimalloc, tbbmalloc, and custom
```

This can be resolved either by defining `HPX_WITH_MALLOC=system` or by installing TCMalloc. This error occurs when users don't specify an option for `HPX_WITH_MALLOC`; in that case, *HPX* will be looking `tcmalloc`, which is the default value.

[186] https://www.cmake.org
[187] https://github.com/STEllAR-GROUP/hpx/issues/5784
[188] https://github.com/STEllAR-GROUP/hpx/issues/5404

Useful suggestions

Reducing compilation time

If you want to significantly reduce compilation time, you can just use the local part of *HPX* for parallelism by disabling the distributed functionality. Moreover, you can avoid compiling examples. These can be done with the following flags:

```
-DHPX_WITH_NETWORKING=OFF
-DHPX_WITH_DISTRIBUTED_RUNTIME=OFF
-DHPX_WITH_EXAMPLES=OFF
-DHPX_WITH_TESTS=OFF
```

Linking *HPX* to your application

If you want to avoid installing and linking *HPX*, you can just build *HPX* and then use the following flag on your *HPX* application CMake configuration:

```
-DHPX_DIR=<build_dir>/lib/cmake/HPX
```

Note: For this to work you need not to specify `-DCMAKE_INSTALL_PREFIX` when building *HPX*.

HPX-application build type conformance

Your application's build type should align with the HPX build type. For example, if you specified `-DCMAKE_BUILD_TYPE=Debug` during the *HPX* compilation, then your application needs to be compiled with the same flag. We recommend keeping a separate build folder for different build types and just point accordingly to the type you want by using `-DHPX_DIR=<build_dir>/lib/cmake/HPX`.

2.4 Terminology

This section gives definitions for some of the terms used throughout the *HPX* documentation and source code.

Locality A locality in *HPX* describes a synchronous domain of execution, or the domain of bounded upper response time. This normally is just a single node in a cluster or a NUMA domain in a SMP machine.

Active Global Address Space

AGAS *HPX* incorporates a global address space. Any executing thread can access any object within the domain of the parallel application with the caveat that it must have appropriate access privileges. The model does not assume that global addresses are cache coherent; all loads and stores will deal directly with the site of the target object. All global addresses within a Synchronous Domain are assumed to be cache coherent for those processor cores that incorporate transparent caches. The Active Global Address Space used by *HPX* differs from research PGAS[189] models. Partitioned Global Address Space is passive in their means of address translation. Copy semantics, distributed compound operations, and affinity relationships are some of the global functionality supported by AGAS.

Process The concept of the "process" in *HPX* is extended beyond that of either sequential execution or communicating sequential processes. While the notion of process suggests action (as do "function" or "subroutine") it has a

[189] https://www.pgas.org/

further responsibility of context, that is, the logical container of program state. It is this aspect of operation that process is employed in *HPX*. Furthermore, referring to "parallel processes" in *HPX* designates the presence of parallelism within the context of a given process, as well as the coarse grained parallelism achieved through concurrency of multiple processes of an executing user job. *HPX* processes provide a hierarchical name space within the framework of the active global address space and support multiple means of internal state access from external sources.

Parcel The Parcel is a component in *HPX* that communicates data, invokes an action at a distance, and distributes flow-control through the migration of continuations. Parcels bridge the gap of asynchrony between synchronous domains while maintaining symmetry of semantics between local and global execution. Parcels enable message-driven computation and may be seen as a form of "active messages". Other important forms of message-driven computation predating active messages include dataflow tokens[190], the J-machine's[191] support for remote method instantiation, and at the coarse grained variations of Unix remote procedure calls, among others. This enables work to be moved to the data as well as performing the more common action of bringing data to the work. A parcel can cause actions to occur remotely and asynchronously, among which are the creation of threads at different system nodes or synchronous domains.

Local Control Object

Lightweight Control Object

LCO A local control object (sometimes called a lightweight control object) is a general term for the synchronization mechanisms used in *HPX*. Any object implementing a certain concept can be seen as an LCO. This concepts encapsulates the ability to be triggered by one or more events which when taking the object into a predefined state will cause a thread to be executed. This could either create a new thread or resume an existing thread.

The LCO is a family of synchronization functions potentially representing many classes of synchronization constructs, each with many possible variations and multiple instances. The LCO is sufficiently general that it can subsume the functionality of conventional synchronization primitives such as spinlocks, mutexes, semaphores, and global barriers. However due to the rich concept an LCO can represent powerful synchronization and control functionality not widely employed, such as dataflow and futures (among others), which open up enormous opportunities for rich diversity of distributed control and operation.

See lcos for more details on how to use LCOs in *HPX*.

Action An action is a function that can be invoked remotely. In *HPX* a plain function can be made into an action using a macro. See applying_actions for details on how to use actions in *HPX*.

Component A component is a C++ object which can be accessed remotely. A component can also contain member functions which can be invoked remotely. These are referred to as component actions. See *Writing components* for details on how to use components in *HPX*.

2.5 Why *HPX*?

Current advances in high performance computing (HPC) continue to suffer from the issues plaguing parallel computation. These issues include, but are not limited to, ease of programming, inability to handle dynamically changing workloads, scalability, and efficient utilization of system resources. Emerging technological trends such as multi-core processors further highlight limitations of existing parallel computation models. To mitigate the aforementioned problems, it is necessary to rethink the approach to parallelization models. ParalleX contains mechanisms such as multi-threading, *parcels*, *global name space* support, percolation and *local control object*s (*LCO*). By design, ParalleX overcomes limitations of current models of parallelism by alleviating contention, latency, overhead and starvation. With ParalleX, it is further possible to increase performance by at least an order of magnitude on challenging parallel algorithms, e.g., dynamic directed graph algorithms and adaptive mesh refinement methods for astrophysics. An additional benefit of ParalleX is fine-grained control of power usage, enabling reductions in power consumption.

[190] http://en.wikipedia.org/wiki/Dataflow_architecture
[191] http://en.wikipedia.org/wiki/J%E2%80%93Machine

2.5.1 ParalleX—a new execution model for future architectures

ParalleX is a new parallel execution model that offers an alternative to the conventional computation models, such as message passing. ParalleX distinguishes itself by:

- Split-phase transaction model
- Message-driven
- Distributed shared memory (not cache coherent)
- Multi-threaded
- Futures synchronization
- *Local Control Object*s (*LCO*s)
- Synchronization for anonymous producer-consumer scenarios
- Percolation (pre-staging of task data)

The ParalleX model is intrinsically latency hiding, delivering an abundance of variable-grained parallelism within a hierarchical namespace environment. The goal of this innovative strategy is to enable future systems delivering very high efficiency, increased scalability and ease of programming. ParalleX can contribute to significant improvements in the design of all levels of computing systems and their usage from application algorithms and their programming languages to system architecture and hardware design together with their supporting compilers and operating system software.

2.5.2 What is *HPX*?

High Performance ParalleX (*HPX*) is the first runtime system implementation of the ParalleX execution model. The *HPX* runtime software package is a modular, feature-complete, and performance-oriented representation of the ParalleX execution model targeted at conventional parallel computing architectures, such as SMP nodes and commodity clusters. It is academically developed and freely available under an open source license. We provide *HPX* to the community for experimentation and application to achieve high efficiency and scalability for dynamic adaptive and irregular computational problems. *HPX* is a C++ library that supports a set of critical mechanisms for dynamic adaptive resource management and lightweight task scheduling within the context of a global address space. It is solidly based on many years of experience in writing highly parallel applications for HPC systems.

The two-decade success of the communicating sequential processes (CSP) execution model and its message passing interface (MPI) programming model have been seriously eroded by challenges of power, processor core complexity, multi-core sockets, and heterogeneous structures of GPUs. Both efficiency and scalability for some current (strong scaled) applications and future Exascale applications demand new techniques to expose new sources of algorithm parallelism and exploit unused resources through adaptive use of runtime information.

The ParalleX execution model replaces CSP to provide a new computing paradigm embodying the governing principles for organizing and conducting highly efficient scalable computations greatly exceeding the capabilities of today's problems. *HPX* is the first practical, reliable, and performance-oriented runtime system incorporating the principal concepts of the ParalleX model publicly provided in open source release form.

HPX is designed by the STE||AR[192] Group (**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch) at Louisiana State University (LSU)[193]'s Center for Computation and Technology (CCT)[194] to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STE||AR[195] is a research group focusing on system software solutions and scientific application development for hybrid and many-core hardware architectures.

[192] https://stellar-group.org
[193] https://www.lsu.edu
[194] https://www.cct.lsu.edu
[195] https://stellar-group.org

For more information about the STE||AR[196] Group, see *People*.

2.5.3 What makes our systems slow?

Estimates say that we currently run our computers at well below 100% efficiency. The theoretical peak performance (usually measured in FLOPS[197]—floating point operations per second) is much higher than any practical peak performance reached by any application. This is particularly true for highly parallel hardware. The more hardware parallelism we provide to an application, the better the application must scale in order to efficiently use all the resources of the machine. Roughly speaking, we distinguish two forms of scalability: strong scaling (see Amdahl's Law[198]) and weak scaling (see Gustafson's Law[199]). Strong scaling is defined as how the solution time varies with the number of processors for a fixed **total** problem size. It gives an estimate of how much faster we can solve a particular problem by throwing more resources at it. Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size **per processor**. In other words, it defines how much more data can we process by using more hardware resources.

In order to utilize as much hardware parallelism as possible an application must exhibit excellent strong and weak scaling characteristics, which requires a high percentage of work executed in parallel, i.e., using multiple threads of execution. Optimally, if you execute an application on a hardware resource with N processors it either runs N times faster or it can handle N times more data. Both cases imply 100% of the work is executed on all available processors in parallel. However, this is just a theoretical limit. Unfortunately, there are more things that limit scalability, mostly inherent to the hardware architectures and the programming models we use. We break these limitations into four fundamental factors that make our systems *SLOW*:

- **S**tarvation occurs when there is insufficient concurrent work available to maintain high utilization of all resources.
- **L**atencies are imposed by the time-distance delay intrinsic to accessing remote resources and services.
- **O**verhead is work required for the management of parallel actions and resources on the critical execution path, which is not necessary in a sequential variant.
- **W**aiting for contention resolution is the delay due to the lack of availability of oversubscribed shared resources.

Each of those four factors manifests itself in multiple and different ways; each of the hardware architectures and programming models expose specific forms. However, the interesting part is that all of them are limiting the scalability of applications no matter what part of the hardware jungle we look at. Hand-helds, PCs, supercomputers, or the cloud, all suffer from the reign of the 4 horsemen: **S**tarvation, **L**atency, **O**verhead, and **C**ontention. This realization is very important as it allows us to derive the criteria for solutions to the scalability problem from first principles, and it allows us to focus our analysis on very concrete patterns and measurable metrics. Moreover, any derived results will be applicable to a wide variety of targets.

2.5.4 Technology demands new response

Today's computer systems are designed based on the initial ideas of John von Neumann[200], as published back in 1945, and later extended by the Harvard architecture[201]. These ideas form the foundation, the execution model, of computer systems we use currently. However, a new response is required in the light of the demands created by today's technology.

So, what are the overarching objectives for designing systems allowing for applications to scale as they should? In our opinion, the main objectives are:

- Performance: as previously mentioned, scalability and efficiency are the main criteria people are interested in.

[196] https://stellar-group.org
[197] http://en.wikipedia.org/wiki/FLOPS
[198] http://en.wikipedia.org/wiki/Amdahl%27s_law
[199] http://en.wikipedia.org/wiki/Gustafson%27s_law
[200] http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf
[201] http://en.wikipedia.org/wiki/Harvard_architecture

- Fault tolerance: the low expected mean time between failures (MTBF[202]) of future systems requires embracing faults, not trying to avoid them.
- Power: minimizing energy consumption is a must as it is one of the major cost factors today, and will continue to rise in the future.
- Generality: any system should be usable for a broad set of use cases.
- Programmability: for programmer this is a very important objective, ensuring long term platform stability and portability.

What needs to be done to meet those objectives, to make applications scale better on tomorrow's architectures? Well, the answer is almost obvious: we need to devise a new execution model—a set of governing principles for the holistic design of future systems—targeted at minimizing the effect of the outlined **SLOW** factors. Everything we create for future systems, every design decision we make, every criteria we apply, have to be validated against this single, uniform metric. This includes changes in the hardware architecture we prevalently use today, and it certainly involves new ways of writing software, starting from the operating system, runtime system, compilers, and at the application level. However, the key point is that all those layers have to be co-designed; they are interdependent and cannot be seen as separate facets. The systems we have today have been evolving for over 50 years now. All layers function in a certain way, relying on the other layers to do so. But we do not have the time to wait another 50 years for a new coherent system to evolve. The new paradigms are needed now—therefore, co-design is the key.

2.5.5 Governing principles applied while developing *HPX*

As it turn out, we do not have to start from scratch. Not everything has to be invented and designed anew. Many of the ideas needed to combat the 4 horsemen already exist, many for more than 30 years. All it takes is to gather them into a coherent approach. We'll highlight some of the derived principles we think to be crucial for defeating **SLOW**. Some of those are focused on high-performance computing, others are more general.

Focus on latency hiding instead of latency avoidance

It is impossible to design a system exposing zero latencies. In an effort to come as close as possible to this goal many optimizations are mainly targeted towards minimizing latencies. Examples for this can be seen everywhere, such as low latency network technologies like InfiniBand[203], caching memory hierarchies in all modern processors, the constant optimization of existing MPI[204] implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use GPGPUs[205] today. It is important to note that existing latencies are often tightly related to some resource having to wait for the operation to be completed. At the same time it would be perfectly fine to do some other, unrelated work in the meantime, allowing the system to hide the latencies by filling the idle-time with useful work. Modern systems already employ similar techniques (pipelined instruction execution in the processor cores, asynchronous input/output operations, and many more). What we propose is to go beyond anything we know today and to make latency hiding an intrinsic concept of the operation of the whole system stack.

[202] http://en.wikipedia.org/wiki/Mean_time_between_failures
[203] http://en.wikipedia.org/wiki/InfiniBand
[204] https://en.wikipedia.org/wiki/Message_Passing_Interface
[205] http://en.wikipedia.org/wiki/GPGPU

Embrace fine-grained parallelism instead of heavyweight threads

If we plan to hide latencies even for very short operations, such as fetching the contents of a memory cell from main memory (if it is not already cached), we need to have very lightweight threads with extremely short context switching times, optimally executable within one cycle. Granted, for mainstream architectures, this is not possible today (even if we already have special machines supporting this mode of operation, such as the Cray XMT[206]). For conventional systems, however, the smaller the overhead of a context switch and the finer the granularity of the threading system, the better will be the overall system utilization and its efficiency. For today's architectures we already see a flurry of libraries providing exactly this type of functionality: non-pre-emptive, task-queue based parallelization solutions, such as Intel Threading Building Blocks (TBB)[207], Microsoft Parallel Patterns Library (PPL)[208], Cilk++[209], and many others. The possibility to suspend a current task if some preconditions for its execution are not met (such as waiting for I/O or the result of a different task), seamlessly switching to any other task which can continue, and to reschedule the initial task after the required result has been calculated, which makes the implementation of latency hiding almost trivial.

Rediscover constraint-based synchronization to replace global barriers

The code we write today is riddled with implicit (and explicit) global barriers. By "global barriers," we mean the synchronization of the control flow between several (very often all) threads (when using OpenMP[210]) or processes (MPI[211]). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP[212] as the system synchronizes the threads used to execute the different iterations in parallel. In MPI[213] each of the communication steps imposes an explicit barrier onto the execution flow as (often all) nodes have to be synchronized. Each of those barriers is like the eye of a needle the overall execution is forced to be squeezed through. Even minimal fluctuations in the execution times of the parallel threads (jobs) causes them to wait. Additionally, it is often only one of the executing threads that performs the actual reduce operation, which further impedes parallelism. A closer analysis of a couple of key algorithms used in science applications reveals that these global barriers are not always necessary. In many cases it is sufficient to synchronize a small subset of the threads. Any operation should proceed whenever the preconditions for its execution are met, and only those. Usually there is no need to wait for iterations of a loop to finish before you can continue calculating other things; all you need is to complete the iterations that produce the required results for the next operation. Good bye global barriers, hello constraint based synchronization! People have been trying to build this type of computing (and even computers) since the 1970s. The theory behind what they did is based on ideas around static and dynamic dataflow. There are certain attempts today to get back to those ideas and to incorporate them with modern architectures. For instance, a lot of work is being done in the area of constructing dataflow-oriented execution trees. Our results show that employing dataflow techniques in combination with the other ideas, as outlined herein, considerably improves scalability for many problems.

[206] http://en.wikipedia.org/wiki/Cray_XMT
[207] https://www.threadingbuildingblocks.org/
[208] https://msdn.microsoft.com/en-us/library/dd492418.aspx
[209] https://software.intel.com/en-us/articles/intel-cilk-plus/
[210] https://openmp.org/wp/
[211] https://en.wikipedia.org/wiki/Message_Passing_Interface
[212] https://openmp.org/wp/
[213] https://en.wikipedia.org/wiki/Message_Passing_Interface

Adaptive locality control instead of static data distribution

While this principle seems to be a given for single desktop or laptop computers (the operating system is your friend), it is everything but ubiquitous on modern supercomputers, which are usually built from a large number of separate nodes (i.e., Beowulf clusters), tightly interconnected by a high-bandwidth, low-latency network. Today's prevalent programming model for those is MPI, which does not directly help with proper data distribution, leaving it to the programmer to decompose the data to all of the nodes the application is running on. There are a couple of specialized languages and programming environments based on PGAS[214] (Partitioned Global Address Space) designed to overcome this limitation, such as Chapel[215], X10[216], UPC[217], or Fortress[218]. However, all systems based on PGAS rely on static data distribution. This works fine as long as this static data distribution does not result in heterogeneous workload distributions or other resource utilization imbalances. In a distributed system these imbalances can be mitigated by migrating part of the application data to different localities (nodes). The only framework supporting (limited) migration today is Charm++[219]. The first attempts towards solving related problem go back decades as well, a good example is the Linda coordination language[220]. Nevertheless, none of the other mentioned systems support data migration today, which forces the users to either rely on static data distribution and live with the related performance hits or to implement everything themselves, which is very tedious and difficult. We believe that the only viable way to flexibly support dynamic and adaptive *locality* control is to provide a global, uniform address space to the applications, even on distributed systems.

Prefer moving work to the data over moving data to the work

For the best performance it seems obvious to minimize the amount of bytes transferred from one part of the system to another. This is true on all levels. At the lowest level we try to take advantage of processor memory caches, thus, minimizing memory latencies. Similarly, we try to amortize the data transfer time to and from GPGPUs[221] as much as possible. At high levels we try to minimize data transfer between different nodes of a cluster or between different virtual machines on the cloud. Our experience (well, it's almost common wisdom) shows that the amount of bytes necessary to encode a certain operation is very often much smaller than the amount of bytes encoding the data the operation is performed upon. Nevertheless, we still often transfer the data to a particular place where we execute the operation just to bring the data back to where it came from afterwards. As an example let's look at the way we usually write our applications for clusters using MPI. This programming model is all about data transfer between nodes. MPI is the prevalent programming model for clusters, and it is fairly straightforward to understand and to use. Therefore, we often write applications in a way that accommodates this model, centered around data transfer. These applications usually work well for smaller problem sizes and for regular data structures. The larger the amount of data we have to churn and the more irregular the problem domain becomes, the worse the overall machine utilization and the (strong) scaling characteristics become. While it is not impossible to implement more dynamic, data driven, and asynchronous applications using MPI, it is somewhat difficult to do so. At the same time, if we look at applications that prefer to execute the code close to the *locality* where the data was placed, i.e., utilizing active messages (for instance based on Charm++[222]), we see better asynchrony, simpler application codes, and improved scaling.

[214] https://www.pgas.org/
[215] https://chapel.cray.com/
[216] https://x10-lang.org/
[217] https://upc.lbl.gov/
[218] https://labs.oracle.com/projects/plrg/Publications/index.html
[219] https://charm.cs.uiuc.edu/
[220] http://en.wikipedia.org/wiki/Linda_(coordination_language)
[221] http://en.wikipedia.org/wiki/GPGPU
[222] https://charm.cs.uiuc.edu/

Favor message driven computation over message passing

Today's prevalently used programming model on parallel (multi-node) systems is MPI. It is based on message passing, as the name implies, which means that the receiver has to be aware of a message about to come in. Both codes, the sender and the receiver, have to synchronize in order to perform the communication step. Even the newer, asynchronous interfaces require explicitly coding the algorithms around the required communication scheme. As a result, everything but the most trivial MPI applications spends a considerable amount of time waiting for incoming messages, thus, causing starvation and latencies to impede full resource utilization. The more complex and more dynamic the data structures and algorithms become, the larger the adverse effects. The community discovered message-driven and data-driven methods of implementing algorithms a long time ago, and systems such as Charm++[223] have already integrated active messages demonstrating the validity of the concept. Message-driven computation allows for sending messages without requiring the receiver to actively wait for them. Any incoming message is handled asynchronously and triggers the encoded action by passing along arguments and—possibly—continuations. *HPX* combines this scheme with work-queue based scheduling as described above, which allows the system to almost completely overlap any communication with useful work, thereby minimizing latencies.

2.6 Additional material

- 2-day workshop held at CSCS in 2016
  - Recorded lectures[224]
  - Slides[225]
- Tutorials repository[226]
- STE||AR Group blog posts[227]
- Basic *HPX* recipes
  - Exporting a free function from a shared library which lives in a namespace, to use as Action[228]
  - Turning a struct or class into a component and use it's methods[229]
  - Creating and referencing components in hpx[230]

2.7 Overview

HPX is organized into different sub-libraries and those in turn into modules. The libraries and modules are independent, with clear dependencies and no cycles. As an end-user, the use of these libraries is completely transparent. If you use e.g. `add_hpx_executable` to create a target in your project you will automatically get all modules as dependencies. See below for a list of the available libraries and modules. Currently these are nothing more than an internal grouping and do not affect usage. They cannot be consumed individually at the moment.

Note: There is a dependency report that displays useful information about the structure of the code. It is available for each commit at *HPX* Dependency report.

223 https://charm.cs.uiuc.edu/
224 https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOmIJ0lFNjKze18
225 https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2016
226 https://github.com/STEllAR-GROUP/tutorials
227 http://stellar-group.org/blog/
228 https://gitlab.com/-/snippets/1821389
229 https://gitlab.com/-/snippets/1822983
230 https://gitlab.com/-/snippets/1828131

2.7.1 Core modules

affinity

The affinity module contains helper functionality for mapping worker threads to hardware resources.

See the API reference of the module for more details.

algorithms

The algorithms module exposes the full set of algorithms defined by the C++ standard. There is also partial support for C++ ranges.

See the *API reference* of the module for more details.

allocator_support

This module provides utilities for allocators. It contains `hpx::util::internal_allocator` which directly forwards allocation calls to `jemalloc`. This utility is is mainly useful on Windows.

See the API reference of the module for more details.

asio

The asio module is a thin wrapper around the Boost.Asio[231] library, providing a few additional helper functions.

See the *API reference* of the module for more details.

assertion

The assertion library implements the macros `HPX_ASSERT` and `HPX_ASSERT_MSG`. Those two macros can be used to implement assertions which are turned of during a release build.

By default, the location and function where the assert has been called from are displayed when the assertion fires. This behavior can be modified by using `hpx::assertion::set_assertion_handler`. When HPX initializes, it uses this function to specify a more elaborate assertion handler. If your application needs to customize this, it needs to do so before calling `hpx::init`, *hpx_main* or using the C-main wrappers.

See the *API reference* of the module for more details.

async_base

The async_base module defines the basic functionality for spawning tasks on thread pools. This module does not implement any functionality on its own, but is extended by *async_local* and *async_distributed* with implementations for the local and distributed cases.

See the *API reference* of this module for more details.

[231] https://www.boost.org/doc/libs/release/doc/html/boost_asio.html

async_combinators

This module contains combinators for futures. The `when_*` functions allow you to turn multiple futures into a single future which is ready when all, any, some, or each of the given futures are ready. The `wait_*` combinators are equivalent to the `when_*` functions except that they do not return a future. Those wait for all futures to become ready before returning to the user. Note that the `wait_*` functions will rethrow one of the exceptions from exceptional futures. The `wait_*_nothrow` combinators are equivalent to the `wait_*` functions exception that they do not throw if one of the futures has become exceptional.

The `split_future` combinator takes a single future of a container (e.g. `tuple`) and turns it into a container of futures.

See *lcos_local*, *synchronization*, and *async_distributed* for other synchronization facilities.

See the *API reference* of this module for more details.

async_cuda

This library adds a simple API that enables the user to retrieve a future from a CUDA[232] stream. Typically, a user may launch one or more kernels and then get a future from the stream that will become ready when those kernels have completed. It is important to note that multiple kernels may be launched without fetching a future, and multiple futures may be obtained from the helper. Please refer to the unit tests and examples for further examples.

See the *API reference* of this module for more details.

async_local

This module extends *async_base* to provide local implementations of `hpx::async`, `hpx::post`, `hpx::sync`, and `hpx::dataflow`. The *async_distributed* module extends the functionality in this module to work with *action*s.

See the API reference of this module for more details.

async_mpi

The MPI library is intended to simplify the process of integrating MPI[233] based codes with the *HPX* runtime. Any MPI function that is asynchronous and uses an `MPI_Request` may be converted into an `hpx::future`. The syntax is designed to allow a simple replacement of the MPI call with a futurized async version that accepts an executor instead of a communicator, and returns a future instead of assigning a request. Typically, an MPI call of the form

```
int MPI_Isend(buf, count, datatype, rank, tag, comm, request);
```

becomes

```
hpx::future<int> f = hpx::async(executor, MPI_Isend, buf, count, datatype, rank, tag);
```

When the MPI operation is complete, the future will become ready. This allows communication to integrated cleanly with the rest of *HPX*, in particular the continuation style of programming may be used to build up more complex code. Consider the following example, that chains user processing, sends and receives using continuations…

```
// create an executor for MPI dispatch
hpx::mpi::experimental::executor exec(MPI_COMM_WORLD);

// post an asynchronous receive using MPI_Irecv
```

(continues on next page)

[232] https://www.nvidia.com/object/cuda_home_new.html
[233] https://en.wikipedia.org/wiki/Message_Passing_Interface

(continued from previous page)

```
hpx::future<int> f_recv = hpx::async(
    exec, MPI_Irecv, &data, rank, MPI_INT, rank_from, i);

// attach a continuation to run when the recv completes,
f_recv.then([=, &tokens, &counter](auto&&)
{
    // call an application specific function
    msg_recv(rank, size, rank_to, rank_from, tokens[i], i);

    // send a new message
    hpx::future<int> f_send = hpx::async(
        exec, MPI_Isend, &tokens[i], 1, MPI_INT, rank_to, i);

    // when that send completes
    f_send.then([=, &tokens, &counter](auto&&)
    {
        // call an application specific function
        msg_send(rank, size, rank_to, rank_from, tokens[i], i);
    });
}
```

The example above makes use of `MPI_Isend` and `MPI_Irecv`, but *any* MPI function that uses requests may be futurized in this manner. The following is a (non exhaustive) list of MPI functions that *should* be supported, though not all have been tested at the time of writing (please report any problems to the issue tracker).

```
int MPI_Isend(...);
int MPI_Ibsend(...);
int MPI_Issend(...);
int MPI_Irsend(...);
int MPI_Irecv(...);
int MPI_Imrecv(...);
int MPI_Ibarrier(...);
int MPI_Ibcast(...);
int MPI_Igather(...);
int MPI_Igatherv(...);
int MPI_Iscatter(...);
int MPI_Iscatterv(...);
int MPI_Iallgather(...);
int MPI_Iallgatherv(...);
int MPI_Ialltoall(...);
int MPI_Ialltoallv(...);
int MPI_Ialltoallw(...);
int MPI_Ireduce(...);
int MPI_Iallreduce(...);
int MPI_Ireduce_scatter(...);
int MPI_Ireduce_scatter_block(...);
int MPI_Iscan(...);
int MPI_Iexscan(...);
int MPI_Ineighbor_allgather(...);
int MPI_Ineighbor_allgatherv(...);
int MPI_Ineighbor_alltoall(...);
int MPI_Ineighbor_alltoallv(...);
```

(continues on next page)

(continued from previous page)

```
int MPI_Ineighbor_alltoallw(...);
```

Note that the *HPX* mpi futurization wrapper should work with *any* asynchronous `MPI` call, as long as the function signature has the last two arguments `MPI_xxx(..., MPI_Comm comm, MPI_Request *request)` - internally these two parameters will be substituted by the executor and future data parameters that are supplied by template instantiations inside the `hpx::mpi` code.

See the API reference of this module for more details.

async_sycl

This module allows creating HPX futures using SYCL[234] events, effectively integrating asynchronous SYCL kernels and memory transfers with HPX. Building on this integration, this module also contains a SYCL executor. This executor encapsulates a SYCL queue. When SYCL queue member functions are launched with this executor, the user can automatically obtain the HPX futures associated with them.

The creation of the HPX futures using SYCL events is based on the same event polling mechanism that the CUDA HPX integration uses. Each registered event gets an associated callback and gets inserted into a callback vector to be polled by the scheduler in between tasks. Once the polling reveals the event is complete, the callback will be called, which in turn sets the future to ready (see sycl_event_callback.cpp). There are multiple adaptions for HipSYCL for this: To keep the runtime alive (avoiding the repeated on-the-fly creation of the runtime during the polling), we keep a default queue. Furthermore, we flush the internal SYCL DAG to ensure that the launched SYCL function is actually being executed.

The SYCL executor offers the usual post and async_execute functions. Additionally, it contains two get_future functions. One expects a pre-existing SYCL event to return a future, the other one does not but will launch an empty SYCL kernel instead, to obtain an event (causing higher overhead for the sake of being more convenient). The post and async_execute implementations here are actually different for HipSYCL and OneAPI, since the sycl::queue in OneAPI uses a different interface (using a code_location parameter) which requires some adaptations here.

To make this module compile, we use the -fno-sycl and -fsycl compiler parameters for the OneAPI use-case (requiring HPX to be compiled with dpcpp). For HipSYCL we use its cmake integration instead (requiring HPX to be compiled with clang++ and including HipSYCL as a library).

To build with OneAPI, use the CMake Variable HPX_WITH_SYCL=ON. To build with HipSYCL, use HPX_WITH_SYCL=ON and HPX_WITH_HIPSYCL=ON (and make sure find_package will find HipSYCL).

Lastly, the module contains three tests/examples. All three implement a simple vector add example. The first one obtains a future using the free method get_future, the second one uses a single SYCL executor and the last one is using multiple executors called from multiple host threads.

To build the tests, use " make tests.unit.modules.async_sycl " To run the tests, use “ctest -R sycl”.

NOTE: Theoretically, this module could work with other SYCL implementations, but was only tested using OneAPI and HipSYCL so far.

See the *API reference* of this module for more details.

[234] https://en.wikipedia.org/wiki/SYCL

batch_environments

This module allows for the detection of execution as batch jobs, a series of programs executed without user intervention. All data is preselected and will be executed according to preset parameters, such as date or completion of another task. Batch environments are especially useful for executing repetitive tasks.

HPX supports the creation of batch jobs through the Portable Batch System (PBS) and SLURM.

For more information on batch environments, see *Running on batch systems* and the API reference for the module.

cache

This module provides two cache data structures:

- *hpx::util::cache::local_cache*
- *hpx::util::cache::lru_cache*

See the *API reference* of the module for more details.

concepts

This module provides helpers for emulating concepts. It provides the following macros:

- HPX_CONCEPT_REQUIRES
- HPX_HAS_MEMBER_XXX_TRAIT_DEF
- HPX_HAS_XXX_TRAIT_DEF

See the API reference of the module for more details.

concurrency

This module provides concurrency primitives useful for multi-threaded programming such as:

- *hpx::barrier*
- hpx::util::cache_line_data and hpx::util::cache_aligned_data: wrappers for aligning and padding data to cache lines.
- various lockfree queue data structures

See the API reference of the module for more details.

config

The config module contains various configuration options, typically hidden behind macros that choose the correct implementation based on the compiler and other available options. It also contains platform independent macros to control inlinining, export sets and more.

See the *API reference* of the module for more details.

config_registry

The config_registry module is a low level module providing helper functionality for registering configuration entries to a global registry from other modules. The `hpx::config_registry::add_module_config` function is used to add configuration options, and `hpx::config_registry::get_module_configs` can be used to retrieve configuration entries registered so far. `add_module_config_helper` can be used to register configuration entries through static global options.

See the API reference of this module for more details.

coroutines

The coroutines module provides coroutine (user-space thread) implementations for different platforms.

See the *API reference* of the module for more details.

datastructures

The datastructures module provides basic data structures (typically provided for compatibility with older C++ standards):

- `hpx::detail::small_vector`
- *`hpx::util::basic_any`*
- `hpx::util::member_pack`
- `hpx::optional`
- *`hpx::tuple`*
- `hpx::variant`

See the *API reference* of the module for more details.

debugging

This module provides helpers for demangling symbol names.

See the *API reference* of the module for more details.

errors

This module provides support for exceptions and error codes:

- *`hpx::exception`*
- *`hpx::error_code`*
- *`hpx::error`*

See the *API reference* of the module for more details.

execution

This library implements executors and execution policies for use with parallel algorithms and other facilities related to managing the execution of tasks.

See the *API reference* of the module for more details.

execution_base

The basic execution module is the main entry point to implement parallel and concurrent operations. It is modeled after P0443[235] with some additions and implementations for the described concepts. Most notably, it provides an abstraction for execution resources, execution contexts and execution agents in such a way, that it provides customization points that those aforementioned concepts can be replaced and combined with ease.

For that purpose, three virtual base classes are provided to be able to provide implementations with different properties:

- **`resource_base`: This is the abstraction for execution resources, that is** for example CPU cores or an accelerator.
- **`context_base`: An execution context uses execution resources and is able** to spawn new execution agents, as new threads of executions on the available resources.
- **`agent_base`: The execution agent represents the thread of execution, and** can be used to yield, suspend, resume or abort a thread of execution.

executors

The executors module exposes executors and execution policies. Most importantly, it exposes the following classes and constants:

- `hpx::execution::sequenced_executor`
- `hpx::execution::parallel_executor`
- `hpx::execution::sequenced_policy`
- `hpx::execution::parallel_policy`
- `hpx::execution::parallel_unsequenced_policy`
- `hpx::execution::sequenced_task_policy`
- `hpx::execution::parallel_task_policy`
- `hpx::execution::seq`
- `hpx::execution::par`
- `hpx::execution::par_unseq`
- `hpx::execution::task`

See the *API reference* of this module for more details.

[235] http://wg21.link/p0443

filesystem

This module provides a compatibility layer for the C++17 filesystem library. If the filesystem library is available this module will simply forward its contents into the `hpx::filesystem` namespace. If the library is not available it will fall back to Boost.Filesystem instead.

See the *API reference* of the module for more details.

format

The format module exposes the `format` and `format_to` functions for formatting strings.

See the API reference of the module for more details.

functional

This module provides function wrappers and helpers for managing functions and their arguments.

- *hpx::function*
- *hpx::function_ref*
- *hpx::move_only_function*
- *hpx::bind*
- *hpx::bind_back*
- *hpx::bind_front*
- `hpx::util::deferred_call`
- *hpx::invoke*
- *hpx::invoke_r*
- *hpx::invoke_fused*
- *hpx::invoke_fused_r*
- *hpx::mem_fn*
- `hpx::util::one_shot`
- `hpx::util::protect`
- `hpx::util::result_of`
- *hpx::placeholders::_1*
- *hpx::placeholders::_2*
- …
- *hpx::placeholders::_9*

See the *API reference* of the module for more details.

futures

This module defines the *hpx::future* and *hpx::shared_future* classes corresponding to the C++ standard library classes std::future[236] and std::shared_future[237]. Note that the specializations of *hpx::future::then* for executors and execution policies are defined in the *execution* module.

See the *API reference* of this module for more details.

hardware

The hardware module abstracts away hardware specific details of timestamps and CPU features.

See the API reference of the module for more details.

hashing

The hashing module provides two hashing implementations:

- `hpx::util::fibhash`
- `hpx::util::jenkins_hash`

See the API reference of the module for more details.

include_local

This module provides no functionality in itself. Instead it provides headers that group together other headers that often appear together. This module provides local-only headers.

See the API reference of this module for more details.

io_service

This module provides an abstraction over Boost.ASIO, combining multiple `asio::io_contexts` into a single pool. *hpx::util::io_service_pool* provides a simple pool of `asio::io_contexts` with an API similar to `asio::io_context`. `hpx::threads::detail::io_service_thread_pool` wraps *hpx::util::io_service_pool* into an interface derived from `hpx::threads::detail::thread_pool_base`.

See the *API reference* of this module for more details.

iterator_support

This module provides helpers for iterators. It provides `hpx::util::iterator_facade` and `hpx::util::iterator_adaptor` for creating new iterators, and the trait `hpx::util::is_iterator` along with more specific iterator traits.

See the API reference of the module for more details.

236 http://en.cppreference.com/w/cpp/thread/future
237 http://en.cppreference.com/w/cpp/thread/shared_future

itt_notify

This module provides support for profiling with Intel VTune[238].

See the API reference of this module for more details.

lci_base

This module provides helper functionality for detecting LCI environments.

See the API reference of this module for more details.

lcos_local

This module provides the following local *LCO*s:

- hpx::lcos::local::and_gate
- hpx::lcos::local::channel
- hpx::lcos::local::one_element_channel
- hpx::lcos::local::receive_channel
- hpx::lcos::local::send_channel
- hpx::lcos::local::guard
- hpx::lcos::local::guard_set
- hpx::lcos::local::run_guarded
- hpx::lcos::local::conditional_trigger
- *hpx::packaged_task*
- *hpx::promise*
- hpx::lcos::local::receive_buffer
- *hpx::lcos::local::trigger*

See *lcos_distributed* for distributed LCOs. Basic synchronization primitives for use in *HPX* threads can be found in *synchronization*. *async_combinators* contains useful utility functions for combining futures.

See the *API reference* of this module for more details.

lock_registration

This module contains fucntionality for registering locks to detect when they are locked and unlocked on different threads.

See the API reference of this module for more details.

[238] https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html

logging

This module provides useful macros for logging information.

See the API reference of the module for more details.

memory

Part of this module is a forked version of `boost::intrusive_ptr` from Boost.SmartPtr[239].

See the API reference of the module for more details.

mpi_base

This module provides helper functionality for detecting MPI[240] environments.

See the API reference of this module for more details.

pack_traversal

This module exposes the basic functionality for traversing various packs, both synchronously and asynchronously: `hpx::util::traverse_pack` and *`hpx::util::traverse_pack_async`*. It also exposes the higher level functionality of unwrapping nested futures: `hpx::util::unwrap` and its function object form `hpx::util::functional::unwrap`.

See the *API reference* of this module for more details.

plugin

This module provides base utilities for creating plugins.

See the API reference of the module for more details.

prefix

This module provides utilities for handling the prefix of an *HPX* application, i.e. the paths used for searching components and plugins.

See the API reference of this module for more details.

preprocessor

This library contains useful preprocessor macros:

- *`HPX_PP_CAT`*: Concatenate two tokens
- *`HPX_PP_EXPAND`*: Expands a preprocessor token
- *`HPX_PP_NARGS`*: Determines the number of arguments passed to a variadic macro
- *`HPX_PP_STRINGIZE`*: Turns a token into a string
- *`HPX_PP_STRIP_PARENS`*: Strips parenthesis from a token

239 https://www.boost.org/doc/libs/release/libs/smart_ptr/doc/html/smart_ptr.html
240 https://en.wikipedia.org/wiki/Message_Passing_Interface

See the *API reference* of the module for more details.

program_options

The module program_options is a direct fork of the Boost.Program_options[241] library (Boost V1.70.0[242]). In order to be included as an *HPX* module, the Boost.Program_options library has been moved to the `namespace hpx::program_options`. We have also replaced all Boost facilities the library depends on with either the equivalent facilities from the standard library or from *HPX*. As a result, the *HPX* program_options module is fully interface compatible with Boost.Program_options (sans the `hpx` namespace and the `#include <hpx/modules/program_options.hpp>` changes that need to be applied to all code relying on this library).

All credit goes to Vladimir Prus, the author of the excellent Boost.Program_options library. All bugs have been introduced by us.

See the API reference of the module for more details.

properties

This module implements the `prefer` customization point for properties in terms of P2220[243]. This differs from P1393[244] in that it relies fully on `tag_invoke` overloads and fewer base customization points. Actual properties are defined in modules. All functionality is experimental and can be accessed through the `hpx::experimental` namespace.

See the API reference of this module for more details.

resiliency

In *HPX*, a program failure is a manifestation of a failing task. This module exposes several APIs that allow users to manage failing tasks in a convenient way by either replaying a failed task or by replicating a specific task.

Task replay is analogous to the Checkpoint/Restart mechanism found in conventional execution models. The key difference being localized fault detection. When the runtime detects an error, it replays the failing task as opposed to completely rolling back the entire program to the previous checkpoint.

Task replication is designed to provide reliability enhancements by replicating a set of tasks and evaluating their results to determine a consensus among them. This technique is most effective in situations where there are few tasks in the critical path of the DAG which leaves the system underutilized or where hardware or software failures may result in an incorrect result instead of an error. However, the drawback of this method is the additional computational cost incurred by repeating a task multiple times.

The following API functions are exposed:

- `hpx::resiliency::experimental::async_replay`: This version of task replay will catch user-defined exceptions and automatically reschedule the task N times before throwing an `hpx::resiliency::experimental::abort_replay_exception` if no task is able to complete execution without an exception.
- `hpx::resiliency::experimental::async_replay_validate`: This version of replay adds an argument to async replay which receives a user-provided validation function to test the result of the task against. If the task's output is validated, the result is returned. If the output fails the check or an exception is thrown, the task is replayed until no errors are encountered or the number of specified retries has been exceeded.

[241] https://www.boost.org/doc/html/program_options.html
[242] https://www.boost.org/doc/libs/1_70_0/doc/html/program_options.html
[243] https://wg21.link/p2220
[244] http://wg21.link/p1393

- `hpx::resiliency::experimental::async_replicate`: This is the most basic implementation of the task replication. The API returns the first result that runs without detecting any errors.
- `hpx::resiliency::experimental::async_replicate_validate`: This API additionally takes a validation function which evaluates the return values produced by the threads. The first task to compute a valid result is returned.
- `hpx::resiliency::experimental::async_replicate_vote`: This API adds a vote function to the basic replicate function. Many hardware or software failures are silent errors which do not interrupt program flow. In order to detect errors of this kind, it is necessary to run the task several times and compare the values returned by every version of the task. In order to determine which return value is "correct", the API allows the user to provide a custom consensus function to properly form a consensus. This voting function then returns the "correct"" answer.
- `hpx::resiliency::experimental::async_replicate_vote_validate`: This combines the features of the previously discussed replicate set. Replicate vote validate allows a user to provide a validation function to filter results. Additionally, as described in replicate vote, the user can provide a "voting function" which returns the consensus formed by the voting logic.
- `hpx::resiliency::experimental::dataflow_replay`: This version of dataflow replay will catch user-defined exceptions and automatically reschedules the task N times before throwing an `hpx::resiliency::experimental::abort_replay_exception` if no task is able to complete execution without an exception. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- `hpx::resiliency::experimental::dataflow_replay_validate` : This version of replay adds an argument to dataflow replay which receives a user-provided validation function to test the result of the task against. If the task's output is validated, the result is returned. If the output fails the check or an exception is thrown, the task is replayed until no errors are encountered or the number of specified retries have been exceeded. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- `hpx::resiliency::experimental::dataflow_replicate`: This is the most basic implementation of the task replication. The API returns the first result that runs without detecting any errors. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- `hpx::resiliency::experimental::dataflow_replicate_validate`: This API additionally takes a validation function which evaluates the return values produced by the threads. The first task to compute a valid result is returned. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- `hpx::resiliency::experimental::dataflow_replicate_vote`: This API adds a vote function to the basic replicate function. Many hardware or software failures are silent errors which do not interrupt program flow. In order to detect errors of this kind, it is necessary to run the task several times and compare the values returned by every version of the task. In order to determine which return value is "correct", the API allows the user to provide a custom consensus function to properly form a consensus. This voting function then returns the "correct" answer. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- `hpx::resiliency::experimental::dataflow_replicate_vote_validate`: This combines the features of the previously discussed replicate set. Replicate vote validate allows a user to provide a validation function to filter results. Additionally, as described in replicate vote, the user can provide a "voting function" which returns the consensus formed by the voting logic. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.

See the *API reference* of the module for more details.

resource_partitioner

The resource_partitioner module defines `hpx::resource::partitioner`, the class used by the runtime and users to partition available hardware resources into thread pools. See *Using the resource partitioner* for more details on using the resource partitioner in applications.

See the API reference of this module for more details.

runtime_configuration

This module handles the configuration options required by the runtime.

See the *API reference* of this module for more details.

schedulers

This module provides schedulers used by thread pools in the *thread_pools* module. There are currently three main schedulers:

- `hpx::threads::policies::local_priority_queue_scheduler`
- `hpx::threads::policies::static_priority_queue_scheduler`
- `hpx::threads::policies::shared_priority_queue_scheduler`

Other schedulers are specializations or variations of the above schedulers. See the examples of the *resource_partitioner* module for examples of specifying a custom scheduler for a thread pool.

See the API reference of this module for more details.

serialization

This module provides serialization primitives and support for all built-in types as well as all C++ Standard Library collection and utility types. This list is extended by *HPX* vocabulary types with proper support for global reference counting. *HPX*'s mode of serialization is derived from Boost's serialization model[245] and, as such, is mostly interface compatible with its Boost counterpart.

The purest form of serializing data is to copy the content of the payload bit by bit; however, this method is impractical for generic C++ types, which might be composed of more than just regular built-in types. Instead, *HPX*'s approach to serialization is derived from the Boost Serialization library, and is geared towards allowing the programmer of a given class explicit control and syntax of what to serialize. It is based on operator overloading of two special archive types that hold a buffer or stream to store the serialized data and is responsible for dispatching the serialization mechanism to the intrusive or non-intrusive version. The serialization process is recursive. Each member that needs to be serialized must be specified explicitly. The advantage of this approach is that the serialization code is written in C++ and leverages all necessary programming techniques. The generic, user-facing interface allows for effective application of the serialization process without obstructing the algorithms that need special code for packing and unpacking. It also allows for optimizations in the implementation of the archives.

See the *API reference* of the module for more details.

[245] https://www.boost.org/doc/libs/1_72_0/libs/serialization/doc/index.html

static_reinit

This module provides a simple wrapper around static variables that can be reinitialized.

See the *API reference* of this module for more details.

string_util

This module contains string utilities inspired by the Boost String Algorithms Library.

See the API reference of this module for more details.

synchronization

This module provides synchronization primitives that should be used rather than the C++ standard ones in *HPX* threads:

- `hpx::barrier`
- `hpx::binary_semaphore`
- `hpx::call_once`
- `hpx::condition_variable`
- `hpx::condition_variable_any`
- `hpx::counting_semaphore`
- `hpx::lcos::local::event`
- `hpx::latch`
- `hpx::mutex`
- `hpx::no_mutex`
- `hpx::once_flag`
- `hpx::recursive_mutex`
- `hpx::shared_mutex`
- `hpx::sliding_semaphore`
- `hpx::spinlock` (*std::mutex* compatible spinlock)
- `hpx::spinlock_no_backoff` (*boost::mutex* compatible spinlock)
- `hpx::spinlock_pool`
- `hpx::stop_callback`
- `hpx::stop_source`
- `hpx::stop_token`
- `hpx::in_place_stop_token`
- `hpx::timed_mutex`
- `hpx::upgrade_to_unique_lock`
- `hpx::upgrade_lock`

See *lcos_local*, *async_combinators*, and *async_distributed* for higher level synchronization facilities.

See the *API reference* of this module for more details.

testing

The testing module contains useful macros for testing. The results of tests can be printed with `hpx::util::report_errors`. The following macros are provided:

- `HPX_TEST`
- `HPX_TEST_MSG`
- `HPX_TEST_EQ`
- `HPX_TEST_NEQ`
- `HPX_TEST_LT`
- `HPX_TEST_LTE`
- `HPX_TEST_RANGE`
- `HPX_TEST_EQ_MSG`
- `HPX_TEST_NEQ_MSG`
- `HPX_SANITY`
- `HPX_SANITY_MSG`
- `HPX_SANITY_EQ`
- `HPX_SANITY_NEQ`
- `HPX_SANITY_LT`
- `HPX_SANITY_LTE`
- `HPX_SANITY_RANGE`
- `HPX_SANITY_EQ_MSG`

See the API reference of the module for more details.

thread_pool_util

This module contains helper functions for asynchronously suspending and resuming thread pools and their worker threads.

See the *API reference* of this module for more details.

thread_pools

This module defines the thread pools and utilities used by the *HPX* runtime. The only thread pool implementation provided by this module is `hpx::threads::detail::scheduled_thread_pool`, which is derived from `hpx::threads::detail::thread_pool_base` defined in the *threading_base* module.

See the API reference of this module for more details.

thread_support

This module provides miscellaneous utilities for threading and concurrency.

See the *API reference* of the module for more details.

threading

This module provides the equivalents of `std::thread` and `std::jthread` for lightweight *HPX* threads:

- `hpx::thread`
- `hpx::jthread`

See the *API reference* of this module for more details.

threading_base

This module contains the base class definition required for threads. The base class `hpx::threads::thread_data` is inherited by two specializations for stackful and stackless threads: `hpx::threads::thread_data_stackful` and `hpx::threads::thread_data_stackless`. In addition, the module defines the base classes for schedulers and thread pools: `hpx::threads::policies::scheduler_base` and `hpx::threads::thread_pool_base`.

See the API reference of this module for more details.

thread_manager

This module defines the `hpx::threads::threadmanager` class. This is used by the runtime to manage the creation and destruction of thread pools. The *resource_partitioner* module handles the partitioning of resources into thread pools, but not the creation of thread pools.

See the API reference of this module for more details.

timed_execution

This module provides extensions to the executor interfaces defined in the *execution* module that allow timed submission of tasks on thread pools (at or after a specified time).

See the *API reference* of this module for more details.

timing

This module provides the timing utilities (clocks and timers).

See the *API reference* of the module for more details.

topology

This module provides the class *hpx::threads::topology* which represents the hardware resources available on a node. The class is a light wrapper around the Portable Hardware Locality (HWLOC)[246] library. The `hpx::threads::cpu_mask` is a small companion class that represents a set of resources on a node.

See the *API reference* of the module for more details.

type_support

This module provides helper facilities related to types.

See the API reference of the module for more details.

util

The util module provides miscellaneous standalone utilities.

See the *API reference* of the module for more details.

version

This module macros and functions for accessing version information about *HPX* and its dependencies.

See the *API reference* of this module for more details.

2.7.2 Main *HPX* modules

actions

TODO: High-level description of the library.

See the *API reference* of this module for more details.

actions_base

TODO: High-level description of the library.

See the *API reference* of this module for more details.

agas

TODO: High-level description of the module.

See the *API reference* of this module for more details.

[246] https://www.open-mpi.org/projects/hwloc/

agas_base

This module holds the implementation of the four AGAS services: primary namespace, locality namespace, component namespace, and symbol namespace.

See the *API reference* of this module for more details.

async_colocated

TODO: High-level description of the module.

See the *API reference* of this module for more details.

async_distributed

This module contains functionality for asynchronously launching work on remote localities: `hpx::async`, `hpx::post`. This module extends the local-only functions in libs_async_local.

See the API reference of this module for more details.

checkpoint

A common need of users is to periodically backup an application. This practice provides resiliency and potential restart points in code. *HPX* utilizes the concept of a `checkpoint` to support this use case.

Found in `hpx/util/checkpoint.hpp`, `checkpoint`s are defined as objects that hold a serialized version of an object or set of objects at a particular moment in time. This representation can be stored in memory for later use or it can be written to disk for storage and/or recovery at a later point. In order to create and fill this object with data, users must use a function called `save_checkpoint`. In code the function looks like this:

```
hpx::future<hpx::util::checkpoint> hpx::util::save_checkpoint(a, b, c, ...);
```

`save_checkpoint` takes arbitrary data containers, such as `int`, `double`, `float`, `vector`, and `future`, and serializes them into a newly created `checkpoint` object. This function returns a `future` to a `checkpoint` containing the data. Here's an example of a simple use case:

```
using hpx::util::checkpoint;
using hpx::util::save_checkpoint;

std::vector<int> vec{1,2,3,4,5};
hpx::future<checkpoint> save_checkpoint(vec);
```

Once the future is ready, the checkpoint object will contain the `vector` `vec` and its five elements.

`prepare_checkpoint` takes arbitrary data containers (same as for `save_checkpoint`), , such as `int`, `double`, `float`, `vector`, and `future`, and calculates the necessary buffer space for the checkpoint that would be created if `save_checkpoint` was called with the same arguments. This function returns a `future` to a `checkpoint` that is appropriately initialized. Here's an example of a simple use case:

```
using hpx::util::checkpoint;
using hpx::util::prepare_checkpoint;

std::vector<int> vec{1,2,3,4,5};
hpx::future<checkpoint> prepare_checkpoint(vec);
```

Once the future is ready, the checkpoint object will be initialized with an appropriately sized internal buffer.

It is also possible to modify the launch policy used by `save_checkpoint`. This is accomplished by passing a launch policy as the first argument. It is important to note that passing `hpx::launch::sync` will cause `save_checkpoint` to return a `checkpoint` instead of a `future` to a `checkpoint`. All other policies passed to `save_checkpoint` will return a `future` to a `checkpoint`.

Sometimes `checkpoint` s must be declared before they are used. `save_checkpoint` allows users to move pre-created `checkpoint` s into the function as long as they are the first container passing into the function (In the case where a launch policy is used, the `checkpoint` will immediately follow the launch policy). An example of these features can be found below:

```
char character = 'd';
int integer = 10;
float flt = 10.01f;
bool boolean = true;
std::string str = "I am a string of characters";
std::vector<char> vec(str.begin(), str.end());
checkpoint archive;

// Test 1
//  test basic functionality
hpx::shared_future<checkpoint> f_archive = save_checkpoint(
    std::move(archive), character, integer, flt, boolean, str, vec);
```

Once users can create `checkpoints` they must now be able to restore the objects they contain into memory. This is accomplished by the function `restore_checkpoint`. This function takes a `checkpoint` and fills its data into the containers it is provided. It is important to remember that the containers must be ordered in the same way they were placed into the `checkpoint`. For clarity see the example below:

```
char character2;
int integer2;
float flt2;
bool boolean2;
std::string str2;
std::vector<char> vec2;

restore_checkpoint(data, character2, integer2, flt2, boolean2, str2, vec2);
```

The core utility of `checkpoint` is in its ability to make certain data persistent. Often, this means that the data needs to be stored in an object, such as a file, for later use. *HPX* has two solutions for these issues: stream operator overloads and access iterators.

HPX contains two stream overloads, `operator<<` and `operator>>`, to stream data out of and into `checkpoint`. Here is an example of the overloads in use below:

```
double a9 = 1.0, b9 = 1.1, c9 = 1.2;
std::ofstream test_file_9("test_file_9.txt");
hpx::future<checkpoint> f_9 = save_checkpoint(a9, b9, c9);
test_file_9 << f_9.get();
test_file_9.close();

double a9_1, b9_1, c9_1;
std::ifstream test_file_9_1("test_file_9.txt");
checkpoint archive9;
```

(continues on next page)

(continued from previous page)

```
    test_file_9_1 >> archive9;
    restore_checkpoint(archive9, a9_1, b9_1, c9_1);
```

This is the primary way to move data into and out of a `checkpoint`. It is important to note, however, that users should be cautious when using a stream operator to load data and another function to remove it (or vice versa). Both `operator<<` and `operator>>` rely on a `.write()` and a `.read()` function respectively. In order to know how much data to read from the `std::istream`, the `operator<<` will write the size of the `checkpoint` before writing the `checkpoint` data. Correspondingly, the `operator>>` will read the size of the stored data before reading the data into a new instance of `checkpoint`. As long as the user employs the `operator<<` and `operator>>` to stream the data, this detail can be ignored.

Important: Be careful when mixing `operator<<` and `operator>>` with other facilities to read and write to a `checkpoint`. `operator<<` writes an extra variable, and `operator>>` reads this variable back separately. Used together the user will not encounter any issues and can safely ignore this detail.

Users may also move the data into and out of a `checkpoint` using the exposed `.begin()` and `.end()` iterators. An example of this use case is illustrated below.

```
    std::ofstream test_file_7("checkpoint_test_file.txt");
    std::vector<float> vec7{1.02f, 1.03f, 1.04f, 1.05f};
    hpx::future<checkpoint> fut_7 = save_checkpoint(vec7);
    checkpoint archive7 = fut_7.get();
    std::copy(archive7.begin(),     // Write data to ofstream
        archive7.end(),             // ie. the file
        std::ostream_iterator<char>(test_file_7));
    test_file_7.close();

    std::vector<float> vec7_1;
    std::vector<char> char_vec;
    std::ifstream test_file_7_1("checkpoint_test_file.txt");
    if (test_file_7_1)
    {
        test_file_7_1.seekg(0, test_file_7_1.end);
        auto length = test_file_7_1.tellg();
        test_file_7_1.seekg(0, test_file_7_1.beg);
        char_vec.resize(length);
        test_file_7_1.read(char_vec.data(), length);
    }
    checkpoint archive7_1(std::move(char_vec));    // Write data to checkpoint
    restore_checkpoint(archive7_1, vec7_1);
```

Checkpointing components

save_checkpoint and restore_checkpoint are also able to store components inside checkpoints. This can be done in one of two ways. First a client of the component can be passed to save_checkpoint. When the user wishes to resurrect the component she can pass a client instance to restore_checkpoint.

This technique is demonstrated below:

```
// Try to checkpoint and restore a component with a client
std::vector<int> vec3{10, 10, 10, 10, 10};

// Create a component instance through client constructor
data_client D(hpx::find_here(), std::move(vec3));
hpx::future<checkpoint> f3 = save_checkpoint(D);

// Create a new client
data_client E;

// Restore server inside client instance
restore_checkpoint(f3.get(), E);
```

The second way a user can save a component is by passing a shared_ptr to the component to save_checkpoint. This component can be resurrected by creating a new instance of the component type and passing a shared_ptr to the new instance to restore_checkpoint.

This technique is demonstrated below:

```
// test checkpoint a component using a shared_ptr
std::vector<int> vec{1, 2, 3, 4, 5};
data_client A(hpx::find_here(), std::move(vec));

// Checkpoint Server
hpx::id_type old_id = A.get_id();

hpx::future<std::shared_ptr<data_server>> f_a_ptr =
    hpx::get_ptr<data_server>(A.get_id());
std::shared_ptr<data_server> a_ptr = f_a_ptr.get();
hpx::future<checkpoint> f = save_checkpoint(a_ptr);
auto&& data = f.get();

// test prepare_checkpoint API
checkpoint c = prepare_checkpoint(hpx::launch::sync, a_ptr);
HPX_TEST(c.size() == data.size());

// Restore Server
// Create a new server instance
std::shared_ptr<data_server> b_server;
restore_checkpoint(data, b_server);
```

checkpoint_base

The checkpoint_base module contains lower level facilities that wrap simple check-pointing capabilities. This module does not implement special handling for futures or components, but simply serializes all arguments to or from a given container.

This module exposes the `hpx::util::save_checkpoint_data`, `hpx::util::restore_checkpoint_data`, and `hpx::util::prepare_checkpoint_data` APIs. These functions encapsulate the basic serialization functionalities necessary to save/restore a variadic list of arguments to/from a given data container.

See the *API reference* of this module for more details.

collectives

The collectives module exposes a set of distributed collective operations. Those can be used to exchange data between participating sites in a coordinated way. At this point the module exposes the following collective primitives:

- `hpx::collectives::all_gather`: receives a set of values from all participating sites.
- `hpx::collectives::all_reduce`: performs a reduction on data from each participating site to each participating site.
- `hpx::collectives::all_to_all`: each participating site provides its element of the data to collect while all participating sites receive the data from every other site.
- `hpx::collectives::broadcast_to` and `hpx::collectives::broadcast_from`: performs a broadcast operation from a root site to all participating sites.
- **cpp:func:hpx::collectives::exclusive_scan** performs an exclusive scan operation

  on a set of values received from all call sites operating on the given base name.
- `hpx::collectives::gather_here` and `hpx::collectives::gather_there`: gathers values from all participating sites.
- **cpp:func:hpx::collectives::inclusive_scan** performs an inclusive scan operation

  on a set of values received from all call sites operating on the given base name.
- `hpx::collectives::reduce_here` and `hpx::collectives::reduce_there`: performs a reduction on data from each participating site to a root site.
- `hpx::collectives::scatter_to` and `hpx::collectives::scatter_from`: receives an element of a set of values operating on the given base name.
- `hpx::lcos::broadcast`: performs a given action on all given global identifiers.
- `hpx::distributed::barrier`: distributed barrier.
- `hpx::lcos::fold`: performs a fold with a given action on all given global identifiers.
- `hpx::distributed::latch`: distributed latch.
- `hpx::lcos::reduce`: performs a reduction on data from each given global identifiers.
- `hpx::lcos::spmd_block`: performs the same operation on a local image while providing handles to the other images.

See the *API reference* of the module for more details.

command_line_handling

The command_line_handling module defines and handles the command-line options required by the *HPX* runtime, combining them with configuration options defined by the *runtime_configuration* module. The actual parsing of command line options is handled by the *program_options* module.

See the API reference of the module for more details.

components

TODO: High-level description of the module.

See the *API reference* of this module for more details.

components_base

TODO: High-level description of the library.

See the *API reference* of this module for more details.

compute

The compute module provides utilities for handling task and memory affinity on host systems.

See the *API reference* of the module for more details.

distribution_policies

TODO: High-level description of the module.

See the *API reference* of this module for more details.

executors_distributed

This module provides the executor `hpx::parallel::execution::disribution_policy_executor`. It allows one to create work that is implicitly distributed over multiple localities.

See the *API reference* of this module for more details.

include

This module provides no functionality in itself. Instead it provides headers that group together other headers that often appear together.

See the API reference of this module for more details.

init_runtime

TODO: High-level description of the library.

See the *API reference* of this module for more details.

lcos_distributed

This module contains distributed *LCO*s. Currently the only LCO provided is :cpp:class::*hpx::lcos::channel*, a construct for sending values from one *locality* to another. See libs_lcos_local for local LCOs.

See the API reference of this module for more details.

naming

TODO: High-level description of the module.

See the API reference of this module for more details.

naming_base

This module provides a forward declaration of *address_type*, *component_type* and *invalid_locality_id*.

See the *API reference* of this module for more details.

parcelport_lci

TODO: High-level description of the module.

See the API reference of this module for more details.

parcelport_libfabric

TODO: High-level description of the module.

See the API reference of this module for more details.

parcelport_mpi

TODO: High-level description of the module.

See the API reference of this module for more details.

parcelport_tcp

TODO: High-level description of the module.

See the API reference of this module for more details.

parcelset

TODO: High-level description of the module.

See the *API reference* of this module for more details.

parcelset_base

TODO: High-level description of the module.

See the *API reference* of this module for more details.

performance_counters

This module provides the basic functionality required for defining performance counters. See *Performance counters* for more information about performance counters.

See the *API reference* of this module for more details.

plugin_factories

TODO: High-level description of the module.

See the *API reference* of this module for more details.

resiliency_distributed

Software resiliency features of *HPX* were introduced in the *resiliency module*. This module extends the APIs to run on distributed-memory systems allowing the user to invoke the failing task on other localities at runtime. This is useful in cases where a node is identified to fail more often (e.g., for certain ALU computes) as the task can now be replayed or replicated among different localities. The API exposed allows for an easy integration with the local only resiliency APIs as well.

Distributed software resilience APIs have a similar function signature and lives under the same namespace of `hpx::resiliency::experimental`. The difference arises in the formal parameters where distributed APIs takes the localities as the first argument, and an action as opposed to a function or a function object. The localities signify the order in which the API will either schedule (in case of Task Replay) tasks in a round robin fashion or replicate the tasks onto the list of localities.

The list of APIs exposed by distributed resiliency modules is the same as those defined in *local resiliency module*.

See the API reference of this module for more details.

runtime_components

TODO: High-level description of the module.

See the *API reference* of this module for more details.

runtime_distributed

TODO: High-level description of the module.

See the *API reference* of this module for more details.

segmented_algorithms

Segmented algorithms extend the usual parallel *algorithms* by providing overloads that work with distributed containers, such as partitioned vectors.

See the *API reference* of the module for more details.

statistics

This module provide some statistics utilities like rolling min/max and histogram.

See the API reference of the module for more details.

2.8 API reference

HPX follows a versioning scheme with three numbers: `major.minor.patch`. We guarantee no breaking changes in the API for patch releases. Minor releases may remove or break existing APIs, but only after a deprecation period of at least two minor releases. In rare cases do we outright remove old and unused functionality without a deprecation period.

We do not provide any ABI compatibility guarantees between any versions, debug and release builds, and builds with different C++ standards.

The public API of *HPX* is presented below. Clicking on a name brings you to the full documentation for the class or function. Including the header specified in a heading brings in the features listed under that heading.

Note: Names listed here are guaranteed stable with respect to semantic versioning. However, at the moment the list is incomplete and certain unlisted features are intended to be in the public API. While we work on completing the list, if you're unsure about whether a particular unlisted name is part of the public API you can get into contact with us or open an issue and we'll clarify the situation.

2.8.1 Public API

Our API is semantically conforming; hence, the reader is highly encouraged to refer to the corresponding facility in the C++ Standard[247] if needed. All names below are also available in the top-level `hpx` namespace unless otherwise noted. The names in `hpx` should be preferred. The names in sub-namespaces will eventually be removed.

[247] https://en.cppreference.com/w/cpp/header

hpx/algorithm.hpp

The header hpx/algorithm.hpp[248] corresponds to the C++ standard library header algorithm[249]. See *Using parallel algorithms* for more information about the parallel algorithms.

Classes

Table 2.123: Classes of header hpx/algorithm.hpp

Class	C++ standard
hpx::experimental::reduction	N4808[250]
hpx::experimental::induction	N4808[251]

Functions

Table 2.124: *hpx* functions of header hpx/algorithm.hpp

hpx function	C++ standard
hpx::adjacent_find	std::adjacent_find[252]
hpx::all_of	std::all_of[253]
hpx::any_of	std::any_of[254]
hpx::copy	std::copy[255]
hpx::copy_if	std::copy_if[256]
hpx::copy_n	std::copy_n[257]
hpx::count	std::count[258]
hpx::count_if	std::count_if[259]
hpx::ends_with	std::ends_with[260]
hpx::equal	std::equal[261]
hpx::fill	std::fill[262]
hpx::fill_n	std::fill_n[263]
hpx::find	std::find[264]
hpx::find_end	std::find_end[265]
hpx::find_first_of	std::find_first_of[266]
hpx::find_if	std::find_if[267]
hpx::find_if_not	std::find_if_not[268]
hpx::for_each	std::for_each[269]
hpx::for_each_n	std::for_each_n[270]
hpx::generate	std::generate[271]
hpx::generate_n	std::generate_n[272]
hpx::includes	std::includes[273]
hpx::inplace_merge	std::inplace_merge[274]
hpx::is_heap	std::is_heap[275]
hpx::is_heap_until	std::is_heap_until[276]
hpx::is_partitioned	std::is_partitioned[277]
hpx::is_sorted	std::is_sorted[278]

continues on next page

[248] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp
[249] http://en.cppreference.com/w/cpp/header/algorithm
[250] http://wg21.link/n4808
[251] http://wg21.link/n4808

Table 2.124 – continued from previous page

hpx function	C++ standard
`hpx::is_sorted_until`	std::is_sorted_until[279]
`hpx::lexicographical_compare`	std::lexicographical_compare[280]
`hpx::make_heap`	std::make_heap[281]
`hpx::max_element`	std::max_element[282]
`hpx::merge`	std::merge[283]
`hpx::min_element`	std::min_element[284]
`hpx::minmax_element`	std::minmax_element[285]
`hpx::mismatch`	std::mismatch[286]
`hpx::move`	std::move[287]
`hpx::none_of`	std::none_of[288]
`hpx::nth_element`	std::nth_element[289]
`hpx::partial_sort`	std::partial_sort[290]
`hpx::partial_sort_copy`	std::partial_sort_copy[291]
`hpx::partition`	std::partition[292]
`hpx::partition_copy`	std::partition_copy[293]
`hpx::experimental::reduce_by_key`	reduce_by_key[294]
`hpx::remove`	std::remove[295]
`hpx::remove_copy`	std::remove_copy[296]
`hpx::remove_copy_if`	std::remove_copy_if[297]
`hpx::remove_if`	std::remove_if[298]
`hpx::replace`	std::replace[299]
`hpx::replace_copy`	std::replace_copy[300]
`hpx::replace_copy_if`	std::replace_copy_if[301]
`hpx::replace_if`	std::replace_if[302]
`hpx::reverse`	std::reverse[303]
`hpx::reverse_copy`	std::reverse_copy[304]
`hpx::rotate`	std::rotate[305]
`hpx::rotate_copy`	std::rotate_copy[306]
`hpx::search`	std::search[307]
`hpx::search_n`	std::search_n[308]
`hpx::set_difference`	std::set_difference[309]
`hpx::set_intersection`	std::set_intersection[310]
`hpx::set_symmetric_difference`	std::set_symmetric_difference[311]
`hpx::set_union`	std::set_union[312]
`hpx::shift_left`	std::shift_left[313]
`hpx::shift_right`	std::shift_right[314]
`hpx::sort`	std::sort[315]
`hpx::experimental::sort_by_key`	sort_by_key[316]
`hpx::stable_partition`	std::stable_partition[317]
`hpx::stable_sort`	std::stable_sort[318]
`hpx::starts_with`	std::starts_with[319]
`hpx::swap_ranges`	std::swap_ranges[320]
`hpx::transform`	std::transform[321]
`hpx::unique`	std::unique[322]
`hpx::unique_copy`	std::unique_copy[323]
`hpx::experimental::for_loop`	N4808[324]
`hpx::experimental::for_loop_strided`	N4808[325]
`hpx::experimental::for_loop_n`	N4808[326]
`hpx::experimental::for_loop_n_strided`	N4808[327]

[252] http://en.cppreference.com/w/cpp/algorithm/adjacent_find
[253] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[254] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[255] http://en.cppreference.com/w/cpp/algorithm/copy
[256] http://en.cppreference.com/w/cpp/algorithm/copy
[257] http://en.cppreference.com/w/cpp/algorithm/copy_n
[258] http://en.cppreference.com/w/cpp/algorithm/count
[259] http://en.cppreference.com/w/cpp/algorithm/count
[260] http://en.cppreference.com/w/cpp/algorithm/ranges/ends_with
[261] http://en.cppreference.com/w/cpp/algorithm/equal
[262] http://en.cppreference.com/w/cpp/algorithm/fill
[263] http://en.cppreference.com/w/cpp/algorithm/fill_n
[264] http://en.cppreference.com/w/cpp/algorithm/find
[265] http://en.cppreference.com/w/cpp/algorithm/find_end
[266] http://en.cppreference.com/w/cpp/algorithm/find_first_of
[267] http://en.cppreference.com/w/cpp/algorithm/find
[268] http://en.cppreference.com/w/cpp/algorithm/find
[269] http://en.cppreference.com/w/cpp/algorithm/for_each
[270] http://en.cppreference.com/w/cpp/algorithm/for_each_n
[271] http://en.cppreference.com/w/cpp/algorithm/generate
[272] http://en.cppreference.com/w/cpp/algorithm/generate_n
[273] http://en.cppreference.com/w/cpp/algorithm/includes
[274] http://en.cppreference.com/w/cpp/algorithm/inplace_merge
[275] http://en.cppreference.com/w/cpp/algorithm/is_heap
[276] http://en.cppreference.com/w/cpp/algorithm/is_heap_until
[277] http://en.cppreference.com/w/cpp/algorithm/is_partitioned
[278] http://en.cppreference.com/w/cpp/algorithm/is_sorted
[279] http://en.cppreference.com/w/cpp/algorithm/is_sorted_until
[280] http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
[281] http://en.cppreference.com/w/cpp/algorithm/make_heap
[282] http://en.cppreference.com/w/cpp/algorithm/max_element
[283] http://en.cppreference.com/w/cpp/algorithm/merge
[284] http://en.cppreference.com/w/cpp/algorithm/min_element
[285] http://en.cppreference.com/w/cpp/algorithm/minmax_element
[286] http://en.cppreference.com/w/cpp/algorithm/mismatch
[287] http://en.cppreference.com/w/cpp/algorithm/move
[288] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[289] http://en.cppreference.com/w/cpp/algorithm/nth_element
[290] http://en.cppreference.com/w/cpp/algorithm/partial_sort
[291] http://en.cppreference.com/w/cpp/algorithm/partial_sort_copy
[292] http://en.cppreference.com/w/cpp/algorithm/partition
[293] http://en.cppreference.com/w/cpp/algorithm/partition_copy
[294] https://thrust.github.io/doc/group__reductions_gad5623f203f9b3fdcab72481c3913f0e0.html
[295] http://en.cppreference.com/w/cpp/algorithm/remove
[296] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[297] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[298] http://en.cppreference.com/w/cpp/algorithm/remove
[299] http://en.cppreference.com/w/cpp/algorithm/replace
[300] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[301] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[302] http://en.cppreference.com/w/cpp/algorithm/replace
[303] http://en.cppreference.com/w/cpp/algorithm/reverse
[304] http://en.cppreference.com/w/cpp/algorithm/reverse_copy
[305] http://en.cppreference.com/w/cpp/algorithm/rotate
[306] http://en.cppreference.com/w/cpp/algorithm/rotate_copy
[307] http://en.cppreference.com/w/cpp/algorithm/search
[308] http://en.cppreference.com/w/cpp/algorithm/search_n
[309] http://en.cppreference.com/w/cpp/algorithm/set_difference
[310] http://en.cppreference.com/w/cpp/algorithm/set_intersection
[311] http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
[312] http://en.cppreference.com/w/cpp/algorithm/set_union
[313] http://en.cppreference.com/w/cpp/algorithm/shift
[314] http://en.cppreference.com/w/cpp/algorithm/shift
[315] http://en.cppreference.com/w/cpp/algorithm/sort
[316] https://thrust.github.io/doc/group__sorting_gabe038d6107f7c824cf74120500ef45ea.html
[317] http://en.cppreference.com/w/cpp/algorithm/stable_partition
[318] http://en.cppreference.com/w/cpp/algorithm/stable_sort
[319] http://en.cppreference.com/w/cpp/algorithm/ranges/starts_with
[320] http://en.cppreference.com/w/cpp/algorithm/swap_ranges
[321] http://en.cppreference.com/w/cpp/algorithm/transform
[322] http://en.cppreference.com/w/cpp/algorithm/unique
[323] http://en.cppreference.com/w/cpp/algorithm/unique_copy
[324] http://wg21.link/n4808
[325] http://wg21.link/n4808

Table 2.125: *hpx::ranges* functions of header hpx/algorithm.hpp

hpx::ranges function	C++ standard
hpx::ranges::adjacent_find	std::adjacent_find[328]
hpx::ranges::all_of	std::all_of[329]
hpx::ranges::any_of	std::any_of[330]
hpx::ranges::copy	std::copy[331]
hpx::ranges::copy_if	std::copy_if[332]
hpx::ranges::copy_n	std::copy_n[333]
hpx::ranges::count	std::count[334]
hpx::ranges::count_if	std::count_if[335]
hpx::ranges::ends_with	std::ends_with[336]
hpx::ranges::equal	std::equal[337]
hpx::ranges::fill	std::fill[338]
hpx::ranges::fill_n	std::fill_n[339]
hpx::ranges::find	std::find[340]
hpx::ranges::find_end	std::find_end[341]
hpx::ranges::find_first_of	std::find_first_of[342]
hpx::ranges::find_if	std::find_if[343]
hpx::ranges::find_if_not	std::find_if_not[344]
hpx::ranges::for_each	std::for_each[345]
hpx::ranges::for_each_n	std::for_each_n[346]
hpx::ranges::generate	std::generate[347]
hpx::ranges::generate_n	std::generate_n[348]
hpx::ranges::includes	std::includes[349]
hpx::ranges::inplace_merge	std::inplace_merge[350]
hpx::ranges::is_heap	std::is_heap[351]
hpx::ranges::is_heap_until	std::is_heap_until[352]
hpx::ranges::is_partitioned	std::is_partitioned[353]
hpx::ranges::is_sorted	std::is_sorted[354]
hpx::ranges::is_sorted_until	std::is_sorted_until[355]
hpx::ranges::make_heap	std::make_heap[356]
hpx::ranges::max_element	std::max_element[357]
hpx::ranges::merge	std::merge[358]
hpx::ranges::min_element	std::min_element[359]
hpx::ranges::minmax_element	std::minmax_element[360]
hpx::ranges::mismatch	std::mismatch[361]
hpx::ranges::move	std::move[362]
hpx::ranges::none_of	std::none_of[363]
hpx::ranges::nth_element	std::nth_element[364]
hpx::ranges::partial_sort	std::partial_sort[365]
hpx::ranges::partial_sort_copy	std::partial_sort_copy[366]
hpx::ranges::partition	std::partition[367]
hpx::ranges::partition_copy	std::partition_copy[368]
hpx::ranges::set_difference	std::set_difference[369]
hpx::ranges::set_intersection	std::set_intersection[370]
hpx::ranges::set_symmetric_difference	std::set_symmetric_difference[371]
hpx::ranges::set_union	std::set_union[372]
hpx::ranges::shift_left	P2440[373]
hpx::ranges::shift_right	P2440[374]
hpx::ranges::sort	std::sort[375]

continues on next page

Table 2.125 – continued from previous page

hpx::ranges function	C++ standard
hpx::ranges::stable_partition	std::stable_partition[376]
hpx::ranges::stable_sort	std::stable_sort[377]
hpx::ranges::starts_with	std::starts_with[378]
hpx::ranges::swap_ranges	std::swap_ranges[379]
hpx::ranges::transform	std::transform[380]
hpx::ranges::unique	std::unique[381]
hpx::ranges::unique_copy	std::unique_copy[382]
hpx::ranges::experimental::for_loop	N4808[383]
hpx::ranges::experimental::for_loop_strided	N4808[384]

hpx/any.hpp

The header hpx/any.hpp[385] corresponds to the C++ standard library header any[386].

hpx::any is compatible with `std::any`.

[328] http://en.cppreference.com/w/cpp/algorithm/ranges/adjacent_find
[329] http://en.cppreference.com/w/cpp/algorithm/ranges/all_any_none_of
[330] http://en.cppreference.com/w/cpp/algorithm/ranges/all_any_none_of
[331] http://en.cppreference.com/w/cpp/algorithm/ranges/copy
[332] http://en.cppreference.com/w/cpp/algorithm/ranges/copy
[333] http://en.cppreference.com/w/cpp/algorithm/ranges/copy_n
[334] http://en.cppreference.com/w/cpp/algorithm/ranges/count
[335] http://en.cppreference.com/w/cpp/algorithm/ranges/count
[336] http://en.cppreference.com/w/cpp/algorithm/ranges/ends_with
[337] http://en.cppreference.com/w/cpp/algorithm/ranges/equal
[338] http://en.cppreference.com/w/cpp/algorithm/ranges/fill
[339] http://en.cppreference.com/w/cpp/algorithm/ranges/fill_n
[340] http://en.cppreference.com/w/cpp/algorithm/ranges/find
[341] http://en.cppreference.com/w/cpp/algorithm/ranges/find_end
[342] http://en.cppreference.com/w/cpp/algorithm/ranges/find_first_of
[343] http://en.cppreference.com/w/cpp/algorithm/ranges/find
[344] http://en.cppreference.com/w/cpp/algorithm/ranges/find
[345] http://en.cppreference.com/w/cpp/algorithm/ranges/for_each
[346] http://en.cppreference.com/w/cpp/algorithm/ranges/for_each_n
[347] http://en.cppreference.com/w/cpp/algorithm/ranges/generate
[348] http://en.cppreference.com/w/cpp/algorithm/ranges/generate_n
[349] http://en.cppreference.com/w/cpp/algorithm/ranges/includes
[350] http://en.cppreference.com/w/cpp/algorithm/ranges/inplace_merge
[351] http://en.cppreference.com/w/cpp/algorithm/ranges/is_heap
[352] http://en.cppreference.com/w/cpp/algorithm/ranges/is_heap_until
[353] http://en.cppreference.com/w/cpp/algorithm/ranges/is_partitioned
[354] http://en.cppreference.com/w/cpp/algorithm/ranges/is_sorted
[355] http://en.cppreference.com/w/cpp/algorithm/ranges/is_sorted_until
[356] http://en.cppreference.com/w/cpp/algorithm/ranges/make_heap
[357] http://en.cppreference.com/w/cpp/algorithm/ranges/max_element
[358] http://en.cppreference.com/w/cpp/algorithm/ranges/merge
[359] http://en.cppreference.com/w/cpp/algorithm/ranges/min_element
[360] http://en.cppreference.com/w/cpp/algorithm/ranges/minmax_element
[361] http://en.cppreference.com/w/cpp/algorithm/ranges/mismatch
[362] http://en.cppreference.com/w/cpp/algorithm/ranges/move
[363] http://en.cppreference.com/w/cpp/algorithm/ranges/all_any_none_of
[364] http://en.cppreference.com/w/cpp/algorithm/ranges/nth_element
[365] http://en.cppreference.com/w/cpp/algorithm/ranges/partial_sort
[366] http://en.cppreference.com/w/cpp/algorithm/ranges/partial_sort_copy
[367] http://en.cppreference.com/w/cpp/algorithm/ranges/partition
[368] http://en.cppreference.com/w/cpp/algorithm/ranges/partition_copy
[369] http://en.cppreference.com/w/cpp/algorithm/ranges/set_difference
[370] http://en.cppreference.com/w/cpp/algorithm/ranges/set_intersection
[371] http://en.cppreference.com/w/cpp/algorithm/ranges/set_symmetric_difference
[372] http://en.cppreference.com/w/cpp/algorithm/ranges/set_union
[373] https://wg21.link/p2440
[374] https://wg21.link/p2440
[375] http://en.cppreference.com/w/cpp/algorithm/ranges/sort
[376] http://en.cppreference.com/w/cpp/algorithm/ranges/stable_partition
[377] http://en.cppreference.com/w/cpp/algorithm/ranges/stable_sort
[378] http://en.cppreference.com/w/cpp/algorithm/ranges/starts_with
[379] http://en.cppreference.com/w/cpp/algorithm/ranges/swap_ranges
[380] http://en.cppreference.com/w/cpp/algorithm/ranges/transform
[381] http://en.cppreference.com/w/cpp/algorithm/ranges/unique
[382] http://en.cppreference.com/w/cpp/algorithm/ranges/unique_copy
[383] http://wg21.link/n4808
[384] http://wg21.link/n4808
[385] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/any.hpp
[386] http://en.cppreference.com/w/cpp/header/any

Classes

Table 2.126: Classes of header hpx/any.hpp

Class	C++ standard
hpx::any	std::any[387]
hpx::any_nonser	
hpx::bad_any_cast	std::bad_any_cast[388]
hpx::unique_any_nonser	

Functions

Table 2.127: Functions of header hpx/any.hpp

Function	C++ standard
hpx::any_cast	std::any_cast[389]
hpx::make_any	std::make_any[390]
hpx::make_any_nonser	
hpx::make_unique_any_nonser	

hpx/assert.hpp

The header hpx/assert.hpp[391] corresponds to the C++ standard library header cassert[392].

HPX_ASSERT is the *HPX* equivalent to `assert` in `cassert`. *HPX_ASSERT* can also be used in CUDA device code.

Macros

Table 2.128: Macros of header hpx/assert.hpp

Macro
HPX_ASSERT
HPX_ASSERT_MSG

[387] http://en.cppreference.com/w/cpp/utility/any
[388] http://en.cppreference.com/w/cpp/utility/any/bad_any_cast
[389] http://en.cppreference.com/w/cpp/utility/any/any_cast
[390] http://en.cppreference.com/w/cpp/utility/any/make_any
[391] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/assertion/include/hpx/assert.hpp
[392] http://en.cppreference.com/w/cpp/header/cassert

hpx/barrier.hpp

The header hpx/barrier.hpp[393] corresponds to the C++ standard library header barrier[394] and contains a distributed barrier implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

Table 2.129: Classes of header `hpx/barrier.hpp`

Class	C++ standard
hpx::barrier	std::barrier[395]

Table 2.130: Distributed implementation of classes of header `hpx/barrier.hpp`

Class
hpx::distributed::barrier

hpx/channel.hpp

The header hpx/channel.hpp[396] contains a local and a distributed channel implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

Table 2.131: Classes of header `hpx/channel.hpp`

Class
`hpx::channel`

Table 2.132: Distributed implementation of classes of header `hpx/channel.hpp`

Class
`hpx::distributed::channel`

[393] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/barrier.hpp
[394] http://en.cppreference.com/w/cpp/header/barrier
[395] http://en.cppreference.com/w/cpp/thread/barrier
[396] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/channel.hpp

hpx/chrono.hpp

The header hpx/chrono.hpp[397] corresponds to the C++ standard library header chrono[398]. The following replacements and extensions are provided compared to chrono[399].

Classes

Table 2.133: Classes of header hpx/chrono.hpp

Class	C++ standard
hpx::chrono::high_resolution_clock	std::high_resolution_clock[400]
hpx::chrono::high_resolution_timer	
hpx::chrono::steady_time_point	std::time_point[401]

hpx/condition_variable.hpp

The header hpx/condition_variable.hpp[402] corresponds to the C++ standard library header condition_variable[403].

Classes

Table 2.134: Classes of header hpx/condition_variable.hpp

Class	C++ standard
hpx::condition_variable	std::condition_variable[404]
hpx::condition_variable_any	std::condition_variable_any[405]
hpx::cv_status	std::cv_status[406]

hpx/exception.hpp

The header hpx/exception.hpp[407] corresponds to the C++ standard library header exception[408]. *hpx::exception* extends `std::exception` and is the base class for all exceptions thrown in *HPX*. *HPX_THROW_EXCEPTION* can be used to throw *HPX* exceptions with file and line information attached to the exception.

[397] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/chrono.hpp
[398] http://en.cppreference.com/w/cpp/header/chrono
[399] http://en.cppreference.com/w/cpp/header/chrono
[400] http://en.cppreference.com/w/cpp/chrono/high_resolution_clock
[401] http://en.cppreference.com/w/cpp/chrono/time_point
[402] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/condition_variable.hpp
[403] http://en.cppreference.com/w/cpp/header/condition_variable
[404] http://en.cppreference.com/w/cpp/thread/condition_variable
[405] http://en.cppreference.com/w/cpp/thread/condition_variable_any
[406] http://en.cppreference.com/w/cpp/thread/cv_status
[407] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/exception.hpp
[408] http://en.cppreference.com/w/cpp/header/exception

Macros

- *HPX_THROW_EXCEPTION*

Classes

Table 2.135: Classes of header `hpx/exception.hpp`

Class	C++ standard
hpx::exception	std::exception[409]

hpx/execution.hpp

The header hpx/execution.hpp[410] corresponds to the C++ standard library header execution[411]. See *High level parallel facilities*, *Using parallel algorithms* and *Executor parameters and executor parameter traits* for more information about execution policies and executor parameters.

Note: These names are only available in the `hpx::execution` namespace, not in the top-level `hpx` namespace.

Constants

Table 2.136: Constants of header `hpx/execution.hpp`

Constant	C++ standard
hpx::execution::seq	std::execution_policy_tag[412]
hpx::execution::par	std::execution_policy_tag[413]
hpx::execution::par_unseq	std::execution_policy_tag[414]
hpx::execution::task	

[409] http://en.cppreference.com/w/cpp/error/exception
[410] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp
[411] http://en.cppreference.com/w/cpp/header/execution
[412] http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag
[413] http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag
[414] http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag

Classes

Table 2.137: Classes of header `hpx/execution.hpp`

Class	C++ standard
hpx::execution::sequenced_policy	std::execution_policy_tag_t[415]
hpx::execution::parallel_policy	std::execution_policy_tag_t[416]
hpx::execution::parallel_unsequenced_policy	std::execution_policy_tag_t[417]
hpx::execution::sequenced_task_policy	
hpx::execution::parallel_task_policy	
hpx::execution::experimental::auto_chunk_size	
hpx::execution::experimental::dynamic_chunk_size	
hpx::execution::experimental::guided_chunk_size	
hpx::execution::experimental::persistent_auto_chunk_size	
hpx::execution::experimental::static_chunk_size	
hpx::execution::experimental::num_cores	

`hpx/functional.hpp`

The header hpx/functional.hpp[418] corresponds to the C++ standard library header functional[419]. *hpx::function* is a more efficient and serializable replacement for `std::function`.

Constants

The following constants correspond to the C++ standard std::placeholders[420]

Table 2.138: Constants of header `hpx/functional.hpp`

Constant
hpx::placeholders::_1
hpx::placeholders::_2
…
hpx::placeholders::_9

[415] http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t
[416] http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t
[417] http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t
[418] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp
[419] http://en.cppreference.com/w/cpp/header/functional
[420] http://en.cppreference.com/w/cpp/utility/functional/placeholders

Classes

Table 2.139: Classes of header `hpx/functional.hpp`

Class	C++ standard
hpx::function	std::function[421]
hpx::function_ref	P0792[422]
hpx::move_only_function	std::move_only_function[423]
hpx::is_bind_expression	std::is_bind_expression[424]
hpx::is_placeholder	std::is_placeholder[425]
hpx::scoped_annotation	

Functions

Table 2.140: Functions of header `hpx/functional.hpp`

Function	C++ standard
hpx::annotated_function	
hpx::bind	std::bind[426]
hpx::bind_back	std::bind_front[427]
hpx::bind_front	std::bind_front[428]
hpx::invoke	std::invoke[429]
hpx::invoke_fused	std::apply[430]
hpx::invoke_fused_r	
hpx::mem_fn	std::mem_fn[431]

hpx/future.hpp

The header hpx/future.hpp[432] corresponds to the C++ standard library header future[433]. See *Extended facilities for futures* for more information about extensions to futures compared to the C++ standard library.

This header file also contains overloads of *hpx::async*, *hpx::post*, *hpx::sync*, and *hpx::dataflow* that can be used with actions. See *Action invocation* for more information about invoking actions.

421 http://en.cppreference.com/w/cpp/utility/functional/function
422 http://wg21.link/p0792
423 http://en.cppreference.com/w/cpp/utility/functional/move_only_function
424 http://en.cppreference.com/w/cpp/utility/functional/is_bind_expression
425 http://en.cppreference.com/w/cpp/utility/functional/is_placeholder
426 http://en.cppreference.com/w/cpp/utility/functional/bind
427 http://en.cppreference.com/w/cpp/utility/functional/bind_front
428 http://en.cppreference.com/w/cpp/utility/functional/bind_front
429 http://en.cppreference.com/w/cpp/utility/functional/invoke
430 http://en.cppreference.com/w/cpp/utility/apply
431 http://en.cppreference.com/w/cpp/utility/functional/mem_fn
432 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp
433 http://en.cppreference.com/w/cpp/header/future

Classes

Table 2.141: Classes of header `hpx/future.hpp`

Class	C++ standard
hpx::future	std::future[434]
hpx::shared_future	std::shared_future[435]
hpx::promise	std::promise[436]
hpx::launch	std::launch[437]
hpx::packaged_task	std::packaged_task[438]

Note: All names except *hpx::promise* are also available in the top-level `hpx` namespace. `hpx::promise` refers to *hpx::distributed::promise*, a distributed variant of *hpx::promise*, but will eventually refer to *hpx::promise* after a deprecation period.

Table 2.142: Distributed implementation of classes of header `hpx/future.hpp`

Class
hpx::distributed::promise

Functions

Table 2.143: Functions of header `hpx/future.hpp`

Function	C++ standard
hpx::async	std::async[439]
hpx::post	
hpx::sync	
hpx::dataflow	
hpx::make_future	
hpx::make_shared_future	
hpx::make_ready_future	P0159[440]
hpx::make_ready_future_alloc	
hpx::make_ready_future_at	
hpx::make_ready_future_after	
hpx::make_exceptional_future	P0159[441]
hpx::when_all	P0159[442]
hpx::when_any	P0159[443]
hpx::when_some	
hpx::when_each	
hpx::wait_all	
hpx::wait_any	
hpx::wait_some	
hpx::wait_each	

[434] http://en.cppreference.com/w/cpp/thread/future
[435] http://en.cppreference.com/w/cpp/thread/shared_future
[436] http://en.cppreference.com/w/cpp/thread/promise
[437] http://en.cppreference.com/w/cpp/thread/launch
[438] http://en.cppreference.com/w/cpp/thread/packaged_task

hpx/init.hpp

The header hpx/init.hpp[444] contains functionality for starting, stopping, suspending, and resuming the *HPX* runtime. This is the main way to explicitly start the *HPX* runtime. See *Starting the HPX runtime* for more details on starting the *HPX* runtime.

Classes

Table 2.144: Classes of header `hpx/init.hpp`

Class
hpx::init_params
hpx::runtime_mode

Functions

Table 2.145: Functions of header `hpx/init.hpp`

Function
hpx::init
hpx::start
hpx::finalize
hpx::disconnect
hpx::suspend
hpx::resume

hpx/latch.hpp

The header hpx/latch.hpp[445] corresponds to the C++ standard library header latch[446]. It contains a local and a distributed latch implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

Table 2.146: Classes of header `hpx/latch.hpp`

Class	C++ standard
hpx::latch	std::latch[447]

[439] http://en.cppreference.com/w/cpp/thread/async
[440] http://wg21.link/p0159
[441] http://wg21.link/p0159
[442] http://wg21.link/p0159
[443] http://wg21.link/p0159
[444] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/init_runtime/include/hpx/init.hpp
[445] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/latch.hpp
[446] http://en.cppreference.com/w/cpp/header/latch
[447] http://en.cppreference.com/w/cpp/thread/latch

Table 2.147: Distributed implementation of classes of header `hpx/latch.hpp`

Class
hpx::distributed::latch

hpx/mutex.hpp

The header hpx/mutex.hpp[448] corresponds to the C++ standard library header mutex[449].

Classes

Table 2.148: Classes of header `hpx/mutex.hpp`

Class	C++ standard
hpx::mutex	std::mutex[450]
hpx::no_mutex	
hpx::once_flag	std::once_flag[451]
hpx::recursive_mutex	std::recursive_mutex[452]
hpx::spinlock	
hpx::timed_mutex	std::timed_mutex[453]
hpx::unlock_guard	

Functions

Table 2.149: Functions of header `hpx/mutex.hpp`

Class	C++ standard
hpx::call_once	std::call_once[454]

hpx/memory.hpp

The header hpx/memory.hpp[455] corresponds to the C++ standard library header memory[456]. It contains parallel versions of the copy, fill, move, and construct helper functions in memory[457]. See *Using parallel algorithms* for more information about the parallel algorithms.

[448] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/mutex.hpp
[449] http://en.cppreference.com/w/cpp/header/mutex
[450] http://en.cppreference.com/w/cpp/thread/mutex
[451] http://en.cppreference.com/w/cpp/thread/once_flag
[452] http://en.cppreference.com/w/cpp/thread/recursive_mutex
[453] http://en.cppreference.com/w/cpp/thread/timed_mutex
[454] http://en.cppreference.com/w/cpp/thread/call_once
[455] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/memory.hpp
[456] http://en.cppreference.com/w/cpp/header/memory
[457] http://en.cppreference.com/w/cpp/header/memory

Functions

Table 2.150: *hpx* functions of header `hpx/memory.hpp`

hpx function	C++ standard
hpx::uninitialized_copy	std::uninitialized_copy[458]
hpx::uninitialized_copy_n	std::uninitialized_copy_n[459]
hpx::uninitialized_default_construct	std::uninitialized_default_construct[460]
hpx::uninitialized_default_construct_n	std::uninitialized_default_construct_n[461]
hpx::uninitialized_fill	std::uninitialized_fill[462]
hpx::uninitialized_fill_n	std::uninitialized_fill_n[463]
hpx::uninitialized_move	std::uninitialized_move[464]
hpx::uninitialized_move_n	std::uninitialized_move_n[465]
hpx::uninitialized_value_construct	std::uninitialized_value_construct[466]
hpx::uninitialized_value_construct_n	std::uninitialized_value_construct_n[467]

Table 2.151: *hpx::ranges* functions of header `hpx/memory.hpp`

hpx::ranges function	C++ standard
hpx::ranges::uninitialized_copy	std::uninitialized_copy[468]
hpx::ranges::uninitialized_copy_n	std::uninitialized_copy_n[469]
hpx::ranges::uninitialized_default_construct	std::uninitialized_default_construct[470]
hpx::ranges::uninitialized_default_construct_n	std::uninitialized_default_construct_n[471]
hpx::ranges::uninitialized_fill	std::uninitialized_fill[472]
hpx::ranges::uninitialized_fill_n	std::uninitialized_fill_n[473]
hpx::ranges::uninitialized_move	std::uninitialized_move[474]
hpx::ranges::uninitialized_move_n	std::uninitialized_move_n[475]
hpx::ranges::uninitialized_value_construct	std::uninitialized_value_construct[476]
hpx::ranges::uninitialized_value_construct_n	std::uninitialized_value_construct_n[477]

[458] http://en.cppreference.com/w/cpp/memory/uninitialized_copy
[459] http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n
[460] http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct
[461] http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n
[462] http://en.cppreference.com/w/cpp/memory/uninitialized_fill
[463] http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n
[464] http://en.cppreference.com/w/cpp/memory/uninitialized_move
[465] http://en.cppreference.com/w/cpp/memory/uninitialized_move_n
[466] http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct
[467] http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct_n
[468] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_copy
[469] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_copy_n
[470] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_default_construct
[471] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_default_construct_n
[472] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_fill
[473] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_fill_n
[474] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_move
[475] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_move_n
[476] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_value_construct
[477] http://en.cppreference.com/w/cpp/memory/ranges/uninitialized_value_construct_n

hpx/numeric.hpp

The header hpx/numeric.hpp[478] corresponds to the C++ standard library header numeric[479]. See *Using parallel algorithms* for more information about the parallel algorithms.

Functions

Table 2.152: *hpx* functions of header hpx/numeric.hpp

hpx function	C++ standard
hpx::adjacent_difference	std::adjacent_difference[480]
hpx::exclusive_scan	std::exclusive_scan[481]
hpx::inclusive_scan	std::inclusive_scan[482]
hpx::reduce	std::reduce[483]
hpx::transform_exclusive_scan	std::transform_exclusive_scan[484]
hpx::transform_inclusive_scan	std::transform_inclusive_scan[485]
hpx::transform_reduce	std::transform_reduce[486]

Table 2.153: *hpx::ranges* functions of header hpx/numeric.hpp

hpx::ranges function
hpx::ranges::adjacent_difference
hpx::ranges::exclusive_scan
hpx::ranges::inclusive_scan
hpx::ranges::reduce
hpx::ranges::transform_exclusive_scan
hpx::ranges::transform_inclusive_scan
hpx::ranges::transform_reduce

hpx/optional.hpp

The header hpx/optional.hpp[487] corresponds to the C++ standard library header optional[488]. `hpx::optional` is compatible with `std::optional`.

[478] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/numeric.hpp
[479] http://en.cppreference.com/w/cpp/header/numeric
[480] http://en.cppreference.com/w/cpp/algorithm/adjacent_difference
[481] http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
[482] http://en.cppreference.com/w/cpp/algorithm/inclusive_scan
[483] http://en.cppreference.com/w/cpp/algorithm/reduce
[484] http://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan
[485] http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan
[486] http://en.cppreference.com/w/cpp/algorithm/transform_reduce
[487] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/optional.hpp
[488] http://en.cppreference.com/w/cpp/header/optional

Constants

- `hpx::nullopt`

Classes

Table 2.154: Classes of header `hpx/optional.hpp`

Class	C++ standard
`hpx::optional`	std::optional[489]
`hpx::nullopt_t`	std::nullopt_t[490]
`hpx::bad_optional_access`	std::bad_optional_access[491]

`hpx/runtime.hpp`

The header hpx/runtime.hpp[492] contains functions for accessing local and distributed runtime information.

Typedefs

Table 2.155: Typedefs of header `hpx/runtime.hpp`

Typedef
hpx::startup_function_type
hpx::shutdown_function_type

Functions

Table 2.156: Functions of header `hpx/runtime.hpp`

Function
hpx::find_root_locality
hpx::find_all_localities
hpx::find_remote_localities
hpx::find_locality
hpx::get_colocation_id
hpx::get_locality_id
hpx::get_num_worker_threads
hpx::get_worker_thread_num
hpx::get_thread_name
hpx::register_pre_startup_function
hpx::register_startup_function
hpx::register_pre_shutdown_function
hpx::register_shutdown_function
hpx::get_num_localities
hpx::get_locality_name

[489] http://en.cppreference.com/w/cpp/utility/optional
[490] http://en.cppreference.com/w/cpp/utility/nullopt_t
[491] http://en.cppreference.com/w/cpp/utility/optional/bad_optional_access
[492] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

hpx/experimental/scope.hpp

The header hpx/experimental/scope.hpp[493] corresponds to the C++ standard library header experimental/scope[494].

Classes

Table 2.157: Classes of header hpx/scope.hpp

Class	C++ standard
hpx::experimental::scope_exit	std::scope_exit[495]
hpx::experimental::scope_fail	std::scope_fail[496]
hpx::experimental::scope_success	std::scope_success[497]

hpx/semaphore.hpp

The header hpx/semaphore.hpp[498] corresponds to the C++ standard library header semaphore[499].

Classes

Table 2.158: Classes of header hpx/semaphore.hpp

Class	C++ standard
hpx::binary_semaphore	std::counting_semaphore[500]
hpx::counting_semaphore	std::counting_semaphore[501]

hpx/shared_mutex.hpp

The header hpx/shared_mutex.hpp[502] corresponds to the C++ standard library header shared_mutex[503].

[493] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/experimental/scope.hpp
[494] http://en.cppreference.com/w/cpp/header/experimental/scope
[495] http://en.cppreference.com/w/cpp/experimental/scope_exit
[496] http://en.cppreference.com/w/cpp/experimental/scope_fail
[497] http://en.cppreference.com/w/cpp/experimental/scope_success
[498] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/semaphore.hpp
[499] http://en.cppreference.com/w/cpp/header/semaphore
[500] http://en.cppreference.com/w/cpp/thread/counting_semaphore
[501] http://en.cppreference.com/w/cpp/thread/counting_semaphore
[502] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/shared_mutex.hpp
[503] http://en.cppreference.com/w/cpp/header/shared_mutex

Classes

Table 2.159: Classes of header `hpx/shared_mutex.hpp`

Class	C++ standard
hpx::shared_mutex	std::shared_mutex[504]

hpx/source_location.hpp

The header hpx/source_location.hpp[505] corresponds to the C++ standard library header source_location[506].

Classes

Table 2.160: Classes of header `hpx/system_error.hpp`

Class	C++ standard
hpx::source_location	std::source_location[507]

hpx/stop_token.hpp

The header hpx/stop_token.hpp[508] corresponds to the C++ standard library header stop_token[509].

Constants

Table 2.161: Constants of header `hpx/stop_token.hpp`

Constant	C++ standard
hpx::nostopstate	std::nostopstate[510]

Classes

Table 2.162: Classes of header `hpx/stop_token.hpp`

Class	C++ standard
hpx::stop_callback	std::stop_callback[511]
hpx::stop_source	std::stop_source[512]
hpx::stop_token	std::stop_token[513]
hpx::nostopstate_t	std::nostopstate_t[514]

504 http://en.cppreference.com/w/cpp/thread/shared_mutex
505 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/source_location.hpp
506 http://en.cppreference.com/w/cpp/header/source_location
507 http://en.cppreference.com/w/cpp/utility/source_location
508 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/stop_token.hpp
509 http://en.cppreference.com/w/cpp/header/stop_token
510 http://en.cppreference.com/w/cpp/thread/stop_source/nostopstate

hpx/system_error.hpp

The header hpx/system_error.hpp[515] corresponds to the C++ standard library header system_error[516].

Classes

Table 2.163: Classes of header hpx/system_error.hpp

Class	C++ standard
hpx::error_code	std::error_code[517]

hpx/task_block.hpp

The header hpx/task_block.hpp[518] corresponds to the task_block feature in N4755[519]. See using_task_block for more details on using task blocks.

Classes

Table 2.164: Classes of header hpx/task_block.hpp

Class
hpx::experimental::task_canceled_exception
hpx::experimental::task_block

Functions

Table 2.165: Functions of header hpx/task_block.hpp

Function
hpx::experimental::define_task_block
hpx::experimental::define_task_block_restore_thread

511 http://en.cppreference.com/w/cpp/thread/stop_callback
512 http://en.cppreference.com/w/cpp/thread/stop_source
513 http://en.cppreference.com/w/cpp/thread/stop_token
514 http://en.cppreference.com/w/cpp/thread/stop_source/nostopstate_t
515 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/system_error.hpp
516 http://en.cppreference.com/w/cpp/header/system_error
517 http://en.cppreference.com/w/cpp/error/error_code
518 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/task_block.hpp
519 http://wg21.link/n4755

hpx/experimental/task_group.hpp

The header hpx/experimental/task_group.hpp[520] corresponds to the task_group feature in oneAPI Threading Building Blocks (oneTBB)[521].

Classes

Table 2.166: Classes of header hpx/experimental/task_group.hpp

Class
hpx::experimental::task_group

hpx/thread.hpp

The header hpx/thread.hpp[522] corresponds to the C++ standard library header thread[523]. The functionality in this header is equivalent to the standard library thread functionality, with the exception that the *HPX* equivalents are implemented on top of lightweight threads and the *HPX* runtime.

Classes

Table 2.167: Classes of header hpx/thread.hpp

Class	C++ standard
hpx::thread	std::thread[524]
hpx::jthread	std::jthread[525]

Functions

Table 2.168: Functions of header hpx/thread.hpp

Function	C++ standard
hpx::this_thread::yield	std::yield[526]
hpx::this_thread::get_id	std::get_id[527]
hpx::this_thread::sleep_for	std::sleep_for[528]
hpx::this_thread::sleep_until	std::sleep_until[529]

[520] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/experimental/task_group.hpp
[521] https://spec.oneapi.io/versions/1.0-rev-3/elements/oneTBB/source/task_scheduler/task_group/task_group_cls.html
[522] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/thread.hpp
[523] http://en.cppreference.com/w/cpp/header/thread
[524] http://en.cppreference.com/w/cpp/thread/thread
[525] http://en.cppreference.com/w/cpp/thread/jthread
[526] http://en.cppreference.com/w/cpp/thread/yield
[527] http://en.cppreference.com/w/cpp/thread/get_id
[528] http://en.cppreference.com/w/cpp/thread/sleep_for
[529] http://en.cppreference.com/w/cpp/thread/sleep_until

hpx/tuple.hpp

The header hpx/tuple.hpp[530] corresponds to the C++ standard library header tuple[531]. *hpx::tuple* can be used in CUDA device code, unlike `std::tuple`.

Constants

Table 2.169: Constants of header `hpx/tuple.hpp`

Constant	C++ standard
hpx::ignore	std::ignore[532]

Classes

Table 2.170: Classes of header `hpx/tuple.hpp`

Class	C++ standard
hpx::tuple	std::tuple[533]
hpx::tuple_size	std::tuple_size[534]
hpx::tuple_element	std::tuple_element[535]

Functions

Table 2.171: Functions of header `hpx/tuple.hpp`

Function	C++ standard
hpx::make_tuple	std::tuple_element[536]
hpx::tie	std::tie[537]
hpx::forward_as_tuple	std::forward_as_tuple[538]
hpx::tuple_cat	std::tuple_cat[539]
hpx::get	std::get[540]

[530] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/tuple.hpp
[531] http://en.cppreference.com/w/cpp/header/tuple
[532] http://en.cppreference.com/w/cpp/utility/tuple/ignore
[533] http://en.cppreference.com/w/cpp/utility/tuple
[534] http://en.cppreference.com/w/cpp/utility/tuple_size
[535] http://en.cppreference.com/w/cpp/utility/tuple_element
[536] http://en.cppreference.com/w/cpp/utility/tuple/tuple_element
[537] http://en.cppreference.com/w/cpp/utility/tuple/tie
[538] http://en.cppreference.com/w/cpp/utility/tuple/forward_as_tuple
[539] http://en.cppreference.com/w/cpp/utility/tuple/tuple_cat
[540] http://en.cppreference.com/w/cpp/utility/tuple/get

hpx/type_traits.hpp

The header hpx/type_traits.hpp[541] corresponds to the C++ standard library header type_traits[542].

Classes

Table 2.172: Classes of header hpx/type_traits.hpp

Class	C++ standard
hpx::is_invocable	std::is_invocable[543]
hpx::is_invocable_r	std::is_invocable[544]

hpx/unwrap.hpp

The header hpx/unwrap.hpp[545] contains utilities for unwrapping futures.

Classes

Table 2.173: Classes of header hpx/unwrap.hpp

Class
hpx::functional::unwrap
hpx::functional::unwrap_n
hpx::functional::unwrap_all

Functions

Table 2.174: Functions of header hpx/unwrap.hpp

Function
hpx::unwrap
hpx::unwrap_n
hpx::unwrap_all
hpx::unwrapping
hpx::unwrapping_n
hpx::unwrapping_all

[541] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/type_traits.hpp
[542] http://en.cppreference.com/w/cpp/header/type_traits
[543] http://en.cppreference.com/w/cpp/types/is_invocable
[544] http://en.cppreference.com/w/cpp/types/is_invocable
[545] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/unwrap.hpp

hpx/version.hpp

The header hpx/version.hpp[546] provides version information about *HPX*.

Macros

Table 2.175: Macros of header hpx/version.hpp

Macro
HPX_VERSION_MAJOR
HPX_VERSION_MINOR
HPX_VERSION_SUBMINOR
HPX_VERSION_FULL
HPX_VERSION_DATE
HPX_VERSION_TAG
HPX_AGAS_VERSION

Functions

Table 2.176: Functions of header hpx/version.hpp

Function
hpx::major_version
hpx::minor_version
hpx::subminor_version
hpx::full_version
hpx::full_version_as_string
hpx::tag
hpx::agas_version
hpx::build_type
hpx::build_date_time

hpx/wrap_main.hpp

The header hpx/wrap_main.hpp[547] does not provide any direct functionality but is used for implicitly using `main` as the runtime entry point. See *Re-use the main() function as the main HPX entry point* for more details on implicitly starting the *HPX* runtime.

[546] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/version/include/hpx/version.hpp
[547] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/wrap/include/hpx/wrap_main.hpp

2.8.2 Public distributed API

Our Public Distributed API offers a rich set of tools and functions that enable developers to harness the full potential of distributed computing. Here, you'll find a comprehensive list of header files, classes and functions for various distributed computing features provided by *HPX*.

hpx/barrier.hpp

The header hpx/barrier.hpp[548] includes a distributed barrier implementation. For information regarding the C++ standard library header barrier[549], see *Public API*.

Classes

Table 2.177: Distributed implementation of classes of header `hpx/barrier.hpp`

Class
hpx::distributed::barrier

Functions

Table 2.178: *hpx* functions of header `hpx/barrier.hpp`

Function
hpx::distributed::wait
hpx::distributed::synchronize

hpx/collectives.hpp

The header hpx/collectives.hpp[550] contains definitions and implementations related to the collectives operations.

Classes

Table 2.179: *hpx* classes of header `hpx/collectives.hpp`

Class
hpx::collectives::num_sites_arg
hpx::collectives::this_site_arg
hpx::collectives::that_site_arg
hpx::collectives::generation_arg
hpx::collectives::root_site_arg
hpx::collectives::tag_arg
hpx::collectives::arity_arg
hpx::collectives::communicator
hpx::collectives::channel_communicator

548 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/barrier.hpp
549 http://en.cppreference.com/w/cpp/header/barrier
550 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/collectives.hpp

Functions

Table 2.180: *hpx* functions of header hpx/collectives.hpp

Function
hpx::collectives::all_gather
hpx::collectives::all_reduce
hpx::collectives::all_to_all
hpx::collectives::broadcast_to
hpx::collectives::broadcast_from
hpx::collectives::create_channel_communicator
hpx::collectives::set
hpx::collectives::get
hpx::collectives::create_communication_set
hpx::collectives::create_communicator
hpx::collectives::create_local_communicator
hpx::collectives::communicator::set_info
hpx::collectives::communicator::get_info
hpx::collectives::communicator::is_root
hpx::collectives::exclusive_scan
hpx::collectives::gather_here
hpx::collectives::gather_there
hpx::collectives::inclusive_scan
hpx::collectives::reduce_here
hpx::collectives::reduce_there
hpx::collectives::scatter_from
hpx::collectives::scatter_to

hpx/latch.hpp

The header hpx/latch.hpp[551] includes a distributed latch implementation. For information regarding the C++ standard library header latch[552], see *Public API*.

[551] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/latch.hpp
[552] http://en.cppreference.com/w/cpp/header/latch

Classes

Table 2.181: Distributed implementation of classes of header `hpx/latch.hpp`

Class
hpx::distributed::latch

Member functions

Table 2.182: *hpx* functions of class `hpx::distributed::latch` from header `hpx/latch.hpp`

Function
hpx::distributed::latch::count_down_and_wait
hpx::distributed::latch::arrive_and_wait
hpx::distributed::latch::count_down
hpx::distributed::latch::is_ready
hpx::distributed::latch::try_wait
hpx::distributed::latch::wait

2.8.3 Full API

The full API of *HPX* is presented below. The listings for the public API above refer to the full documentation below.

Note: Most names listed in the full API reference are implementation details or considered unstable. They are listed mostly for completeness. If there is a particular feature you think deserves being in the public API we may consider promoting it. In general we prioritize making sure features corresponding to C++ standard library features are stable and complete.

algorithms

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::experimental::task_canceled_exception, hpx::experimental::task_block, hpx::experimental::define_task_block, hpx::experimental::define_task_block_restore_thread

Defined in header hpx/task_block.hpp[553].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[553] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/task_block.hpp

namespace **experimental**

Top-level namespace.

Functions

template<typename **ExPolicy**, typename **F**>
decltype(auto) **define_task_block**(*ExPolicy* &&policy, *F* &&f)

Constructs a task_block, *tr*, using the given execution policy *policy*,and invokes the expression *f(tr)* on the user-provided object, *f*.

Postcondition: All tasks spawned from *f* have finished execution. A call to define_task_block may return on a different thread than that on which it was called.

Note: It is expected (but not mandated) that f will (directly or indirectly) call tr.run(*callable_object*).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the task block may be parallelized.
- **F** – The type of the user defined function to invoke inside the define_task_block (deduced). *F* shall be MoveConstructible.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **f** – The user defined function to invoke inside the task block. Given an lvalue *tr* of type task_block, the expression, (void)f(tr), shall be well-formed.

Throws *exception_list* – specified in Exception Handling.

template<typename **F**>
void **define_task_block**(*F* &&f)

Constructs a task_block, tr, and invokes the expression *f(tr)* on the user-provided object, *f*. This version uses *parallel_policy* for task scheduling.

Postcondition: All tasks spawned from *f* have finished execution. A call to define_task_block may return on a different thread than that on which it was called.

Note: It is expected (but not mandated) that f will (directly or indirectly) call tr.run(*callable_object*).

Template Parameters **F** – The type of the user defined function to invoke inside the define_task_block (deduced). *F* shall be MoveConstructible.

Parameters **f** – The user defined function to invoke inside the task block. Given an lvalue *tr* of type task_block, the expression, (void)f(tr), shall be well-formed.

Throws *exception_list* – specified in Exception Handling.

template<typename **ExPolicy**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **define_task_block_restore_thread**(*ExPolicy* &&policy, *F* &&f)

Constructs a task_block, tr, and invokes the expression *f(tr)* on the user-provided object, *f*.

Postcondition: All tasks spawned from *f* have finished execution. A call to *define_task_block_restore_thread* always returns on the same thread as that on which it was called.

Note: It is expected (but not mandated) that f will (directly or indirectly) call tr.run(*callable_object*).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the task block may be parallelized.
- **F** – The type of the user defined function to invoke inside the define_task_block (deduced). *F* shall be MoveConstructible.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **f** – The user defined function to invoke inside the define_task_block. Given an lvalue *tr* of type task_block, the expression, (void)f(tr), shall be well-formed.

Throws *exception_list* – specified in Exception Handling.

template<typename **F**>
void **define_task_block_restore_thread**(*F* &&f)

Constructs a task_block, tr, and invokes the expression *f(tr)* on the user-provided object, *f*. This version uses *parallel_policy* for task scheduling.

Postcondition: All tasks spawned from *f* have finished execution. A call to *define_task_block_restore_thread* always returns on the same thread as that on which it was called.

Note: It is expected (but not mandated) that f will (directly or indirectly) call tr.run(*callable_object*).

Template Parameters **F** – The type of the user defined function to invoke inside the define_task_block (deduced). *F* shall be MoveConstructible.

Parameters **f** – The user defined function to invoke inside the define_task_block. Given an lvalue *tr* of type task_block, the expression, (void)f(tr), shall be well-formed.

Throws *exception_list* – specified in Exception Handling.

template<typename **ExPolicy** = *hpx*::*execution*::*parallel_policy*>

class **task_block**

#include <task_block.hpp> The class *task_block* defines an interface for forking and joining parallel tasks. The *define_task_block* and *define_task_block_restore_thread* function templates create an object of type *task_block* and pass a reference to that object to a user-provided callable object.

An object of class task_block cannot be constructed, destroyed, copied, or moved except by the implementation of the task region library. Taking the address of a *task_block* object via operator& or addressof is ill formed. The result of obtaining its address by any other means is unspecified.

A task_block is active if it was created by the nearest enclosing task block, where "task block" refers to an invocation of define_task_block or define_task_block_restore_thread and "nearest

enclosing" means the most recent invocation that has not yet completed. Code designated for execution in another thread by means other than the facilities in this section (e.g., using thread or async) are not enclosed in the task region and a *task_block* passed to (or captured by) such code is not active within that code. Performing any operation on a *task_block* that is not active results in undefined behavior.

The task_block that is active before a specific call to the run member function is not active within the asynchronous function that invoked run. (The invoked function should not, therefore, capture the task_block from the surrounding block.)

```
Example:
   define_task_block([&](auto& tr) {
       tr.run([&] {
           tr.run([] { f(); });               // Error: tr is not active
           define_task_block([&](auto& tr) { // Nested task block
               tr.run(f);                     // OK: inner tr is active
               /// ...
           });
       }); /// ...
   });
```

Template Parameters **ExPolicy** – The execution policy an instance of a task_block was created with. This defaults to *parallel_policy*.

Public Types

using **execution_policy** = *ExPolicy*

Refers to the type of the execution policy used to create the task_block.

Public Functions

inline constexpr *execution_policy* const &**get_execution_policy**() const noexcept

Return the execution policy instance used to create this task_block

template<typename **F**, typename ...**Ts**>
inline void **run**(*F* &&f, *Ts*&&... ts)

Causes the expression f() to be invoked asynchronously. The invocation of f is permitted to run on an unspecified thread in an unordered fashion relative to the sequence of operations following the call to run(f) (the continuation), or indeterminately sequenced within the same thread as the continuation.

The call to *run* synchronizes with the invocation of f. The completion of f() synchronizes with the next invocation of wait on the same *task_block* or completion of the nearest enclosing task block (i.e., the *define_task_block* or *define_task_block_restore_thread* that created this task block).

Requires: F shall be MoveConstructible. The expression, (void)f(), shall be well-formed.

Precondition: this shall be the active *task_block*.

Postconditions: A call to run may return on a different thread than that on which it was called.

Note: The call to *run* is sequenced before the continuation as if *run* returns on the same thread. The invocation of the user-supplied callable object f may be immediate or may be delayed until compute resources are available. *run* might or might not return before invocation of f completes.

Throws `task_canceled_exception` – described in Exception Handling.

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline void **run**(*Executor* &&exec, *F* &&f, *Ts*&&... ts)

Causes the expression f() to be invoked asynchronously using the given executor. The invocation of f is permitted to run on an unspecified thread associated with the given executor and in an unordered fashion relative to the sequence of operations following the call to run(exec, f) (the continuation), or indeterminately sequenced within the same thread as the continuation.

The call to *run* synchronizes with the invocation of f. The completion of f() synchronizes with the next invocation of wait on the same *task_block* or completion of the nearest enclosing task block (i.e., the *define_task_block* or *define_task_block_restore_thread* that created this task block).

Requires: Executor shall be a type modeling the Executor concept. F shall be MoveConstructible. The expression, (void)f(), shall be well-formed.

Precondition: this shall be the active *task_block*.

Postconditions: A call to run may return on a different thread than that on which it was called.

Note: The call to *run* is sequenced before the continuation as if *run* returns on the same thread. The invocation of the user-supplied callable object f may be immediate or may be delayed until compute resources are available. *run* might or might not return before invocation of f completes.

Throws `task_canceled_exception` – described in Exception Handling. The function will also throw an *exception_list* holding all exceptions that were caught while executing the tasks.

inline void **wait**()

Blocks until the tasks spawned using this *task_block* have finished.

Precondition: this shall be the active *task_block*.

Postcondition: All tasks spawned by the nearest enclosing task region have finished. A call to wait may return on a different thread than that on which it was called.

```
Example:
   define_task_block([&](auto& tr) {
       tr.run([&]{ process(a, w, x); }); // Process a[w] through a[x]
       if (y < x) tr.wait();   // Wait if overlap between [w, x) and [y,␣
↪z)
       process(a, y, z);       // Process a[y] through a[z]
   });
```

Note: The call to *wait* is sequenced before the continuation as if *wait* returns on the same thread.

Throws `This` – function may throw task_canceled_exception, as described in Exception Handling. The function will also throw a *exception_list* holding all exceptions that were caught while executing the tasks.

inline *ExPolicy* &**policy**() noexcept

Returns a reference to the execution policy used to construct this object.

Precondition: this shall be the active *task_block*.

inline constexpr *ExPolicy* const &**policy**() const noexcept

Returns a reference to the execution policy used to construct this object.

Precondition: this shall be the active *task_block*.

Private Members

hpx::*experimental*::*task_group* **tasks_**

threads::thread_id_type **id_**

ExPolicy **policy_**

class **task_canceled_exception** : public *exception*

#include <task_block.hpp> The class task_canceled_exception defines the type of objects thrown by *task_block::run* or *task_block::wait* if they detect that an exception is pending within the current parallel region.

Public Functions

inline **task_canceled_exception**() noexcept

namespace **parallel**

Typedefs

typedef *hpx*::*experimental*::*task_canceled_exception* **instead**

Functions

```
template<typename ExPolicy, typename F>  HPX_DEPRECATED_V (1, 9,
"hpx::parallel:v2::define_task_block is deprecated,
use " "hpx::experimental::define_task_block instead") hpx
```

```
template<typename ExPolicy, typename F>  HPX_DEPRECATED_V (1, 9,
"hpx::parallel:v2::define_task_block is deprecated,
use " "hpx::experimental::define_task_block instead") void define_task_block(ExPolicy &&policy
```

```
template<typename F>  HPX_DEPRECATED_V (1, 9,
"hpx::parallel:v2::define_task_block is deprecated,
use " "hpx::experimental::define_task_block instead") void define_task_block(F &&f)
```

Variables

F && f {returnhpx::experimental::define_task_block(policy, f)

hpx::experimental::task_group

Defined in header hpx/experimental/task_group.hpp[554].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Typedefs

using **instead** = *hpx*::*experimental*::*task_group*

namespace **experimental**

Top-level namespace.

class **task_group**

#include <task_group.hpp> A `task_group` represents concurrent execution of a group of tasks. Tasks can be dynamically added to the group while it is executing.

Public Functions

task_group()

~task_group()

task_group(*task_group* const&) = delete

task_group(*task_group*&&) = delete

task_group &**operator=**(*task_group* const&) = delete

task_group &**operator=**(*task_group*&&) = delete

template<typename **Executor**, typename **F**, typename ...**Ts**>

[554] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/experimental/task_group.hpp

inline void **run**(*Executor* &&exec, *F* &&f, *Ts*&&... ts)

Adds a task to compute `f()` and returns immediately.

Template Parameters

- **Executor** – The type of the executor to associate with this execution policy.
- **F** – The type of the user defined function to invoke.
- **Ts** – The type of additional arguments used to invoke `f()`.

Parameters

- **exec** – The executor to use for the execution of the parallel algorithm the returned execution policy is used with.
- **f** – The user defined function to invoke inside the task group.
- **ts** – Additional arguments to use to invoke `f()`.

template<typename **F**, typename ...**Ts**>
inline void **run**(*F* &&f, *Ts*&&... ts)

Adds a task to compute `f()` and returns immediately.

Template Parameters

- **F** – The type of the user defined function to invoke.
- **Ts** – The type of additional arguments used to invoke `f()`.

Parameters

- **f** – The user defined function to invoke inside the task group.
- **ts** – Additional arguments to use to invoke `f()`.

void **wait**()

Waits for all tasks in the group to complete or be cancelled.

void **add_exception**(*std*::exception_ptr p)

Adds an exception to this `task_group`.

Private Types

using **shared_state_type** = *lcos*::detail::future_data<void>

Private Functions

void **serialize**(*serialization*::output_archive&, unsigned const)

Private Members

hpx::*lcos*::*local*::*latch* **latch_**

hpx::intrusive_ptr<*shared_state_type*> **state_**

hpx::*exception_list* **errors_**

std::atomic<bool> **has_arrived_**

Private Static Functions

static inline constexpr void **serialize**(*serialization*::input_archive&, unsigned const) noexcept

Friends

friend class serialization::access

hpx::adjacent_difference

Defined in header hpx/algorithm.hpp[555].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter1**, typename **FwdIter2**>
FwdIter2 **adjacent_difference**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Assigns each value in the range given by result its corresponding element in the range [first, last] and the one preceding it except *result, which is assigned *first.

Note: Complexity: Exactly (last - first) - 1 application of the binary operator and (last - first) assignments.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the input range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the sequence of elements the results will be assigned to.

Returns The *adjacent_difference* algorithm returns a *FwdIter2*. The *adjacent_difference* algorithm returns an iterator to the element past the last element written.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>

[555] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **adjacent_difference**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Assigns each value in the range given by result its corresponding element in the range [first, last] and the one preceding it except *result, which is assigned *first. Executed according to the policy.

The difference operations in the parallel *adjacent_difference* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The difference operations in the parallel *adjacent_difference* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly (last - first) - 1 application of the binary operator and (last - first) assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the input range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the sequence of elements the results will be assigned to.

Returns The *adjacent_difference* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *adjacent_difference* algorithm returns an iterator to the element past the last element written.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Op**>
FwdIter2 **adjacent_difference**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op)

Assigns each value in the range given by result its corresponding element in the range [first, last] and the one preceding it except *result, which is assigned *first

Note: Complexity: Exactly (last - first) - 1 application of the binary operator and (last - first) assignments.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the input range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Op** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *adjacent_difference* requires *Op* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the sequence of elements the results will be assigned to.
- **op** – The binary operator which returns the difference of elements. The signature should be equivalent to the following:

```
bool op(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter1* can be dereferenced and then implicitly converted to the dereferenced type of *dest*.

Returns The *adjacent_difference* algorithm returns *FwdIter2*. The *adjacent_difference* algorithm returns an iterator to the element past the last element written.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **adjacent_difference**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op)

Assigns each value in the range given by result its corresponding element in the range [first, last] and the one preceding it except *result, which is assigned *first

The difference operations in the parallel *adjacent_difference* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The difference operations in the parallel *adjacent_difference* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly (last - first) - 1 application of the binary operator and (last - first) assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **FwdIter1** – The type of the source iterators used for the input range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Op** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *adjacent_difference* requires *Op* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the sequence of elements the results will be assigned to.
- **op** – The binary operator which returns the difference of elements. The signature should be equivalent to the following:

```
bool op(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter1* can be dereferenced and then implicitly converted to the dereferenced type of *dest*.

Returns The *adjacent_difference* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *adjacent_difference* algorithm returns an iterator to the element past the last element written.

hpx::adjacent_find

Defined in header hpx/algorithm.hpp[556].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **Pred** = *hpx*::*parallel*::detail::equal_to>
InIter **adjacent_find**(*InIter* first, *InIter* last, *Pred* &&pred = *Pred*())

Searches the range [first, last) for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (*result* - *first*) + 1 and (*last* - *first*) - 1 application of the predicate where *result* is the value returned

Template Parameters

[556] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **InIter** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1* .

Returns The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = *hpx*::*parallel*::detail::equal_to>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **adjacent_find**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Pred* &&pred = *Pred*())

Searches the range [first, last) for two consecutive identical elements. This version uses the given binary predicate pred

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *adjacent_find* is available if the user decides to provide their algorithm their own binary predicate *pred*.

Note: Complexity: Exactly the smaller of (*result* - *first*) + 1 and (*last* - *first*) - 1 application of the predicate where *result* is the value returned

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of a forward iterator.

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .

Returns The *adjacent_find* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

hpx::all_of, hpx::any_of, hpx::none_of

Defined in header hpx/algorithm.hpp[557].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>
util::detail::algorithm_result_t<*ExPolicy*, bool> **none_of**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Checks if unary predicate *f* returns true for no elements in the range [first, last).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *last* - *first* applications of the predicate *f*

[557] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *none_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

template<typename **InIter**, typename **F**>
bool **none_of**(*InIter* first, *InIter* last, *F* &&f)

Checks if unary predicate *f* returns true for no elements in the range [first, last).

Note: Complexity: At most *last* - *first* applications of the predicate *f*

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

Returns The *none_of* algorithm returns a *bool* . The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>
util::detail::algorithm_result_t<*ExPolicy*, bool> **any_of**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Checks if unary predicate *f* returns true for at least one element in the range [first, last).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *last - first* applications of the predicate *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *any_of* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *any_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

template<typename **InIter**, typename **F**>
bool **any_of**(*InIter* first, *InIter* last, *F* &&f)

Checks if unary predicate *f* returns true for at least one element in the range [first, last).

Note: Complexity: At most *last* - *first* applications of the predicate *f*

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *any_of* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

Returns The *any_of* algorithm returns a *bool* . The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>
util::detail::algorithm_result_t<*ExPolicy*, bool> **all_of**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Checks if unary predicate *f* returns true for all elements in the range [first, last).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *last* - *first* applications of the predicate *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *all_of* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *all_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

template<typename **ExPolicy**, typename **InIter**, typename **F**>
bool **all_of**(*InIter* first, *InIter* last, *F* &&f)

Checks if unary predicate *f* returns true for all elements in the range [first, last).

Note: Complexity: At most *last* - *first* applications of the predicate *f*

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *all_of* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

Returns The *all_of* algorithm returns a *bool* . The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

hpx::copy, hpx::copy_n, hpx::copy_if

Defined in header hpx/algorithm.hpp[558].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Executed according to the policy.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

[558] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- `first` – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last` – Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest` – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns a *hpx::future<FwdIter2> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2>* otherwise. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **FwdIter2**>
FwdIter2 **copy**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- `FwdIter1` – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2` – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- `first` – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last` – Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest` – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns a *FwdIter2* . The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **copy_n**(*ExPolicy* &&policy, *FwdIter1* first, *Size* count, *FwdIter2* dest)

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. Executed according to the policy.

The assignments in the parallel *copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy_n* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *copy_n* algorithm returns Iterator in the destination range, pointing past the last element copied if count>0 or result otherwise.

template<typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
FwdIter2 **copy_n**(*FwdIter1* first, *Size* count, *FwdIter2* dest)

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy_n* algorithm returns a *FwdIter2* . The *copy_n* algorithm returns Iterator in the destination range, pointing past the last element copied if count>0 or result otherwise.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **copy_if**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Pred* &&pred)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the predicate *f* returns true. The order of the elements that are not removed is preserved. Executed according to the policy.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *f*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.

Returns The *copy_if* algorithm returns a *hpx::future<FwdIter2>* > if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *copy_if* algorithm returns output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>
FwdIter2 **copy_if**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Pred* &&pred)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the predicate *f* returns true. The order of the elements that are not removed is preserved.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *f*.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.

Returns The *copy_if* algorithm returns a *FwdIter2* . The *copy_if* algorithm returns output iterator to the element in the destination range, one past the last element copied.

hpx::count, hpx::count_if

Defined in header hpx/algorithm.hpp[559].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[559] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **T**>
util::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<*FwdIter*>::difference_type>::type **count**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* const &value)

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*. Executed according to the policy.

The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* comparisons.

Note: The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **FwdIter** – The type of the source iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to search for (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.

Returns The *count* algorithm returns a *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIterB>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **InIter**, typename **T**>

std::iterator_traits<*InIter*>::difference_type **count**(*InIter* first, *InIter* last, *T* const &value)

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

Note: Complexity: Performs exactly *last* - *first* comparisons.

Template Parameters

- **InIter** – The type of the source iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to search for (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.

Returns The *count* algorithm returns a *difference_type* (where *difference_type* is defined by *std::iterator_traits<InIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>
util::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<*FwdIter*>::difference_type>::type **count_if**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true. Executed according to the policy.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Note: The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *count_if* algorithm returns *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **InIter**, typename **F**>
std::iterator_traits<*InIter*>::difference_type **count_if**(*InIter* first, *InIter* last, *F* &&f)

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **InIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

Returns The *count_if* algorithm returns *difference_type* (where a difference_type is defined by *std::iterator_traits<InIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

hpx::destroy, hpx::destroy_n

Defined in header hpx/algorithm.hpp[560].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter**>
util::detail::algorithm_result_t<*ExPolicy*> **destroy**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last). Executed according to the policy.

The operations in the parallel *destroy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *destroy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* operations.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

[560] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Returns The *destroy* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **FwdIter**>
void **destroy**(*FwdIter* first, *FwdIter* last)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

Note: Complexity: Performs exactly *last - first* operations.

Template Parameters **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *destroy* algorithm returns a *void*

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **destroy_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, first + count). Executed according to the policy.

The operations in the parallel *destroy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *destroy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* operations, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply this algorithm to.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *destroy_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **FwdIter**, typename **Size**>
FwdIter **destroy_n**(*FwdIter* first, *Size* count)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, first + count).

Note: Complexity: Performs exactly *count* operations, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply this algorithm to.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *destroy_n* algorithm returns a *FwdIter* . The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

hpx::ends_with

Defined in header hpx/algorithm.hpp[561].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter1**, typename **InIter2**, typename **Pred**>
bool **ends_with**(*InIter1* first1, *InIter1* last1, *InIter2* first2, *InIter2* last2, *Pred* &&pred)

Checks whether the second range defined by [first1, last1) matches the suffix of the first range defined by [first2, last2)

The assignments in the parallel *ends_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

[561] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Template Parameters

- **InIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- **Pred** – The binary predicate that compares the projected elements.

Parameters

- **first1** – Refers to the beginning of the source range.
- **last1** – Refers to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.

Returns The *ends_with* algorithm returns *bool*. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, bool>::type **ends_with**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&pred)

Checks whether the second range defined by [first1, last1) matches the suffix of the first range defined by [first2, last2). Executed according to the policy.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The binary predicate that compares the projected elements.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

- **first1** – Refers to the beginning of the source range.
- **last1** – Refers to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for

Returns The *ends_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

hpx::equal

Defined in header hpx/algorithm.hpp[562].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
util::detail::algorithm_result_t<*ExPolicy*, bool> **equal**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op = *Pred*())

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise. Executed according to the policy.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(min(*last1* - *first1*, *last2* - *first2*)) applications of the predicate *op*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

[562] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *equal* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
util::detail::algorithm_result_t<*ExPolicy*, bool> **equal**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2)

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise. Executed according to policy.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(min(*last1* - *first1*, *last2* - *first2*)) applications of the predicate *std::equal_to*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`FwdIter1`** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **`FwdIter2`** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first1`** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **`last1`** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **`first2`** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **`last2`** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.

Returns The *equal* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
util::detail::algorithm_result_t<*ExPolicy*, bool> **equal**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *Pred* &&op = *Pred*())

Returns true if the range [first1, last1) is equal to the range starting at first2, and false otherwise. Executed according to policy.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(min(*last1* - *first1*, *last2* - *first2*)) applications of the predicate *op*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *equal* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
util::detail::algorithm_result_t<*ExPolicy*, bool> **equal**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2)

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise. Executed according to policy.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *last1* - *first1* applications of the predicate *op*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.

Returns The *equal* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
bool **equal**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op = *Pred*())

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *op*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.

- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *equal* algorithm returns a *bool* . The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **FwdIter1**, typename **FwdIter2**>
bool **equal**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2)

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *std::equal_to*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.

- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.

Returns The *equal* algorithm returns a *bool* . The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
bool **equal**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *Pred* &&op = *Pred*())

Returns true if the range [first1, last1) is equal to the range [first2, first2 + (last1 - first1)), and false otherwise.

Note: Complexity: At most *last1* - *first1* applications of the predicate *op*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *equal* algorithm returns a *bool* . The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

hpx::exclusive_scan

Defined in header hpx/algorithm.hpp[563].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **OutIter**, typename **T**>
OutIter **exclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, …, *(first + (i - result) - 1))

The reduce operations in the parallel *exclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *std::plus<T>*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

[563] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.

Returns The *exclusive_scan* algorithm returns *OutIter*. The *exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **exclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, ..., *(first + (i - result) - 1))

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *std::plus*<*T*>.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.

Returns The *exclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **OutIter**, typename **T**, typename **Op**>
OutIter **exclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest, *T* init, *Op* &&op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, …, *(first + (i - result) - 1)).

The reduce operations in the parallel *exclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Op** – The type of the binary function object used for the reduction operation.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.

- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *exclusive_scan* algorithm returns *OutIter*. The *exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**, typename **T**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **exclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *T* init, *Op* &&op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, …, *(first + (i - result) - 1)).

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Op** – The type of the binary function object used for the reduction operation.

- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *exclusive_scan* algorithm returns a *hpx::future<OutIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *OutIter* otherwise. The *exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::fill, hpx::fill_n

Defined in header hpx/algorithm.hpp[564].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **T**>
util::detail::algorithm_result_t<*ExPolicy*> **fill**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* value)

Assigns the given value to the elements in the range [first, last). Executed according to the policy.

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

[564] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

template<typename **FwdIter**, typename **T**>
void **fill**(*FwdIter* first, *FwdIter* last, *T* value)

Assigns the given value to the elements in the range [first, last).

> **Note:** Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns a *void*.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **T**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **fill_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *T* value)

Assigns the given value value to the first count elements in the range beginning at first if count > 0. Does nothing otherwise. Executed according to the policy.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, for count > 0.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

template<typename **FwdIter**, typename **Size**, typename **T**>
FwdIter **fill_n**(*FwdIter* first, *Size* count, *T* value)

Assigns the given value value to the first count elements in the range beginning at first if count > 0. Does nothing otherwise.

Note: Complexity: Performs exactly *count* assignments, for count > 0.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill_n* algorithm returns a *FwdIter*.

hpx::find, hpx::find_if, hpx::find_if_not, hpx::find_end, hpx::find_first_of

Defined in header hpx/algorithm.hpp[565].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **T**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **find**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* const &val)

Returns the first element in the range [first, last) that is equal to value. Executed according to the policy.

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to find (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **val** – the value to compare the elements to

Returns The *find* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

template<typename **InIter**, typename **T**>

[565] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

InIter **find**(*InIter* first, *InIter* last, *T* const &val)

Returns the first element in the range [first, last) that is equal to value. Executed according to the policy.

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **InIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to find (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **val** – the value to compare the elements to

Returns The *find* algorithm returns a *InIter*. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **find_if**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Returns the first element in the range [first, last) for which predicate *f* returns true. Executed according to the policy.

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **f** – The unary predicate which returns true for the required element. The signature of the predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Returns The *find_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if* algorithm returns the first element in the range [first,last) that satisfies the predicate *f*. If no such element exists that satisfies the predicate f, the algorithm returns *last*.

template<typename **InIter**, typename **F**>
InIter **find_if**(*InIter* first, *InIter* last, *F* &&f)

Returns the first element in the range [first, last) for which predicate *f* returns true.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **InIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **f** – The unary predicate which returns true for the required element. The signature of the predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type*.

Returns The *find_if* algorithm returns a *InIter*. The *find_if* algorithm returns the first element in the range [first,last) that satisfies the predicate *f*. If no such element exists that satisfies the predicate f, the algorithm returns *last*.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>

util::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **find_if_not**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Returns the first element in the range [first, last) for which predicate *f* returns false. Executed according to the policy.

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **f** – The unary predicate which returns false for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Returns The *find_if_not* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if_not* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate f, the algorithm returns *last*.

template<typename **FwdIter**, typename **F**>
FwdIter **find_if_not**(*FwdIter* first, *FwdIter* last, *F* &&f)

Returns the first element in the range [first, last) for which predicate *f* returns false.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **f** – The unary predicate which returns false for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

Returns The *find_if_not* algorithm returns a *FwdIter*. The *find_if_not* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate f, the algorithm returns *last*.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter1*> **find_end**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op = *Pred*())

Returns the last subsequence of elements [first2, last2) found in the range [first, last) using the given predicate *op* to compare elements. Executed according to the policy.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **last2** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.

Returns The *find_end* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_end* algorithm returns an iterator to the beginning of the last subsequence [first2, last2) in range [first, last). If the length of the subsequence [first2, last2) is greater than the length of the range [first1, last1), *last1* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last1* is also returned.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter1*> **find_end**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2)

Returns the last subsequence of elements [first2, last2) found in the range [first, last). Elements are compared using *operator==*. Executed according to the policy.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **last2** – Refers to the end of the sequence of elements of the algorithm will be searching for.

Returns The *find_end* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_end* algorithm returns an iterator to the beginning of the last subsequence [first2, last2) in range [first, last). If the length of the subsequence [first2, last2) is greater than the length of the range [first1, last1), *last1* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last1* is also returned.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
FwdIter1 **find_end**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op = *Pred*())

Returns the last subsequence of elements [first2, last2) found in the range [first, last) using the given predicate *op* to compare elements.

This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **last2** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.

Returns The *find_end* algorithm returns a *FwdIter1*. The *find_end* algorithm returns an iterator to the beginning of the last subsequence [first2, last2) in range [first, last). If the length of the subsequence [first2, last2) is greater than the length of the range [first1, last1), *last1* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last1* is also returned.

template<typename **FwdIter1**, typename **FwdIter2**>
FwdIter1 **find_end**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2)

Returns the last subsequence of elements [first2, last2) found in the range [first, last). Elements are compared using *operator==*.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.

- **first2** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **last2** – Refers to the end of the sequence of elements of the algorithm will be searching for.

Returns The *find_end* algorithm returns a *FwdIter1*. The *find_end* algorithm returns an iterator to the beginning of the last subsequence [first2, last2) in range [first, last). If the length of the subsequence [first2, last2) is greater than the length of the range [first1, last1), *last1* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last1* is also returned.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter1*> **find_first_of**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* s_first, *FwdIter2* s_last, *Pred* &&op = *Pred*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses binary predicate *op* to compare elements. Executed according to the policy.

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.

- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.

Returns The *find_first_of* algorithm returns a *hpx::future<FwdIter1>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter1* otherwise. The *find_first_of* algorithm returns an iterator to the first element in the range [first, last) that is equal to an element from the range [s_first, s_last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last* is also returned.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter1*> **find_first_of**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* s_first, *FwdIter2* s_last)

Searches the range [first, last) for any elements in the range [s_first, s_last). Elements are compared using *operator==*. Executed according to the policy.

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.

Returns The *find_first_of* algorithm returns a *hpx::future<FwdIter1>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter1* otherwise. The *find_first_of* algorithm returns an iterator to the first element in the range [first, last) that is equal to an element from the range [s_first, s_last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last* is also returned.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
FwdIter1 **find_first_of**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* s_first, *FwdIter2* s_last, *Pred* &&op = *Pred*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses binary predicate *op* to compare elements.

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.

- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.

Returns The *find_first_of* algorithm returns a *FwdIter1*. The *find_first_of* algorithm returns an iterator to the first element in the range [first, last) that is equal to an element from the range [s_first, s_last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last* is also returned.

template<typename **FwdIter1**, typename **FwdIter2**>
FwdIter1 **find_first_of**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* s_first, *FwdIter2* s_last)

Searches the range [first, last) for any elements in the range [s_first, s_last). Elements are compared using *operator==*.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.

Returns The *find_first_of* algorithm returns a *FwdIter1*. The *find_first_of* algorithm returns an iterator to the first element in the range [first, last) that is equal to an element from the range [s_first, s_last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *last* is also returned.

hpx::for_each, hpx::for_each_n

Defined in header hpx/algorithm.hpp[566].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **F**>
F **for_each**(*InIter* first, *InIter* last, *F* &&f)

Applies *f* to the result of dereferencing every iterator in the range [first, last).

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Note: Complexity: Applies *f* exactly *last* - *first* times.

Template Parameters

- **InIter** – The type of the source begin and end iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). F must meet requirements of *MoveConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

Returns *f*.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>
util::detail::algorithm_result_t<*ExPolicy*, void> **for_each**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Applies *f* to the result of dereferencing every iterator in the range [first, last). Executed according to the policy.

[566] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Applies *f* exactly *last* - *first* times.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter** – The type of the source begin and end iterator used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *for_each* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns void otherwise.

template<typename **InIter**, typename **Size**, typename **F**>
InIter **for_each_n**(*InIter* first, *Size* count, *F* &&f)

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Note: Complexity: Applies *f* exactly *count* times.

Template Parameters

- **InIter** – The type of the source begin and end iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **F** – The type of the function/function object to use (deduced). F must meet requirements of *MoveConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  <ignored> pred(const Type &a);
  ```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

Returns *first* + *count* for non-negative values of *count* and *first* for negative values.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **F**>
util::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **for_each_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *F* &&f)

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1. Executed according to the policy.

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each_n* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Applies *f* exactly *count* times.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each_n* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *for_each_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *first* + *count* for non-negative values of *count* and *first* for negative values.

hpx::experimental::for_loop, hpx::experimental::for_loop_strided, hpx::experimental::for_loop_n, hpx::experimental::for_loop_n_strided

Defined in header hpx/algorithm.hpp[567].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **experimental**

Top-level namespace.

[567] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **I**, typename ...**Args**>
void **for_loop**(*std*::decay_t<*I*> first, *I* last, *Args*&&... args)

The for_loop implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last - first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction

objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

template<typename **ExPolicy**, typename **I**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **for_loop**(*ExPolicy* &&policy, *std*::decay_t<*I*> first, *I* last, *Args*&&... args)

The for_loop implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator. Executed according to the policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last - first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.

- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

template<typename **I**, typename **S**, typename ...**Args**>
void **for_loop_strided**(*std*::decay_t<*I*> first, *I* last, *S* stride, *Args*&&... args)

The for_loop_strided implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last* - *first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **stride** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

  ```
  <ignored> pred(I const& a, ...);
  ```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

template<typename **ExPolicy**, typename **I**, typename **S**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **for_loop_strided**(*ExPolicy* &&policy, *std*::decay_t<*I*> first, *I* last, *S* stride, *Args*&&... args)

The for_loop_strided implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator. Executed according to the policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last* - *first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **stride** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

  ```
  <ignored> pred(I const& a, ...);
  ```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop_strided* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

template<typename **I**, typename **Size**, typename ...**Args**>
void **for_loop_n**(*I* first, *Size* size, *Args*&&... args)

The for_loop_n implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop_n without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last* - *first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **size** – Refers to the number of items the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

template<typename **ExPolicy**, typename **I**, typename **Size**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **for_loop_n**(*ExPolicy* &&policy, *I* first, *Size* size, *Args*&&... args)

The for_loop_n implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator. Executed according to the policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last* - *first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.

- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **size** – Refers to the number of items the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

template<typename **I**, typename **Size**, typename **S**, typename ...**Args**>
void **for_loop_n_strided**(*I* first, *Size* size, *S* stride, *Args*&&... args)

The for_loop_n_strided implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last - first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **size** – Refers to the number of items the algorithm will be applied to.
- **stride** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

template<typename **ExPolicy**, typename **I**, typename **Size**, typename **S**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **for_loop_n_strided**(*ExPolicy* &&policy, *I* first, *Size* size, *S* stride, *Args*&&... args)

The for_loop_n_strided implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator. Executed according to the policy.

Requires: *I* shall be an integral type or meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of *MoveConstructible*.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is *last - first*.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **I** – The type of the iteration variable. This could be an (forward) iterator type or an integral type.
- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **size** – Refers to the number of items the algorithm will be applied to.
- **stride** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(I const& a, ...);
```

The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop_n_strided* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

hpx::experimental::induction

Defined in header hpx/algorithm.hpp[568].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **experimental**

Top-level namespace.

Functions

template<typename **T**>
constexpr *hpx*::*parallel*::detail::induction_stride_helper<*T*> **induction**(*T* &&value, *std*::size_t stride)

The function template returns an induction object of unspecified type having a value type and encapsulating an initial value *value* of that type and, optionally, a stride.

For each element in the input range, a looping algorithm over input sequence *S* computes an induction value from an induction variable and ordinal position *p* within *S* by the formula i + p * stride if a stride was specified or i + p otherwise. This induction value is passed to the element access function.

If the *value* argument to *induction* is a non-const lvalue, then that lvalue becomes the live-out object for the returned induction object. For each induction object that has a live-out object, the looping algorithm assigns the value of i + n * stride to the live-out object upon return, where *n* is the number of elements in the input range.

Template Parameters `T` – The value type to be used by the induction object.

Parameters

- `value` – [in] The initial value to use for the induction object
- `stride` – [in] The (optional) stride to use for the induction object (default: 1)

Returns This returns an induction object with value type *T*, initial value *value*, and (if specified) stride *stride*. If *T* is an lvalue of non-const type, *value* is used as the live-out object for the induction object; otherwise there is no live-out object.

namespace **parallel**

hpx::experimental::reduction

Defined in header hpx/algorithm.hpp[569].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **experimental**

Top-level namespace.

[568] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp
[569] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **T**, typename **Op**>
constexpr *hpx*::*parallel*::detail::reduction_helper<*T*, *std*::decay_t<*Op*>> **reduction**(*T* &var, *T* const &identity, *Op* &&combiner)

The function template returns a reduction object of unspecified type having a value type and encapsulating an identity value for the reduction, a combiner function object, and a live-out object from which the initial value is obtained and into which the final value is stored.

A parallel algorithm uses reduction objects by allocating an unspecified number of instances, called views, of the reduction's value type. Each view is initialized with the reduction object's identity value, except that the live-out object (which was initialized by the caller) comprises one of the views. The algorithm passes a reference to a view to each application of an element-access function, ensuring that no two concurrently-executing invocations share the same view. A view can be shared between two applications that do not execute concurrently, but initialization is performed only once per view.

Modifications to the view by the application of element access functions accumulate as partial results. At some point before the algorithm returns, the partial results are combined, two at a time, using the reduction object's combiner operation until a single value remains, which is then assigned back to the live-out object.

T shall meet the requirements of *CopyConstructible* and *MoveAssignable*. The expression

```
var = combiner(var, var)
```

shall be well formed.

Note: In order to produce useful results, modifications to the view should be limited to commutative operations closely related to the combiner operation. For example if the combiner is plus<T>, incrementing the view would be consistent with the combiner but doubling it or assigning to it would not.

Template Parameters
- **T** – The value type to be used by the induction object.
- **Op** – The type of the binary function (object) used to perform the reduction operation.

Parameters
- **var** – [in,out] The life-out value to use for the reduction object. This will hold the reduced value after the algorithm is finished executing.
- **identity** – [in] The identity value to use for the reduction operation.
- **combiner** – [in] The binary function (object) used to perform a pairwise reduction on the elements.

Returns This returns a reduction object of unspecified type having a value type of *T*. When the return value is used by an algorithm, the reference to *var* is used as the live-out object, new views are initialized to a copy of identity, and views are combined by invoking the copy of combiner, passing it the two views to be combined.

namespace **parallel**

hpx::generate, hpx::generate_n

Defined in header hpx/algorithm.hpp[570].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **generate**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Assign each element in range [*first*, *last*) a value generated by the given function object *f*. Executed according to the policy.

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

  The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

[570] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Returns The *generate* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise.

template<typename **FwdIter**, typename **F**>
FwdIter **generate**(*FwdIter* first, *FwdIter* last, *F* &&f)

Assign each element in range [*first*, *last*) a value generated by the given function object *f*.

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

  The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

Returns The *generate* algorithm returns a *FwdIter*.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **generate_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *F* &&f)

Assigns each element in range [*first*, *first*+*)* a value generated by the given function object *f*. Executed according to the policy.

The assignments in the parallel *generate_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *generate_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *count* invocations of *f* and assignments, for count > 0.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements in the sequence the algorithm will be applied to.
- **f** – Refers to the generator function object that will be called. The signature of the function should be equivalent to

```
Ret fun();
```

The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *generate_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. *generate_n* algorithm returns iterator one past the last element assigned if count>0, first otherwise.

template<typename **FwdIter**, typename **Size**, typename **F**>
FwdIter **generate_n**(*FwdIter* first, *Size* count, *F* &&f)

Assigns each element in range [*first*, *first+)* a value generated by the given function object *f*.

Note: Complexity: Exactly *count* invocations of *f* and assignments, for count > 0.

Template Parameters

- **Size** – The type of a non-negative integral value specifying the number of items to iterate over.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements in the sequence the algorithm will be applied to.
- **f** – Refers to the generator function object that will be called. The signature of the function should be equivalent to

```
Ret fun();
```

The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *generate_n* algorithm returns a *FwdIter*. *generate_n* algorithm returns iterator one past the last element assigned if count>0, first otherwise.

hpx::includes

Defined in header hpx/algorithm.hpp[571].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool>::type **includes**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op = *Pred*())

Returns true if every element from the sorted range [*first2*, *last2*) is found within the sorted range [*first1*, *last1*). Also returns true if [*first2*, *last2*) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*. Executed according to the policy.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: At most 2*(N1+N2-1) comparisons, where N1 = std::distance(first1, last1) and N2 = std::distance(first2, last2).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *includes* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

[571] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *includes* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *includes* algorithm returns true every element from the sorted range [*first2*, *last2*) is found within the sorted range [*first1*, *last1*). Also returns true if [*first2*, *last2*) is empty.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = *hpx*::*parallel*::detail::less>
bool **includes**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op = *Pred*())

Returns true if every element from the sorted range [*first2*, *last2*) is found within the sorted range [*first1*, *last1*). Also returns true if [*first2*, *last2*) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*.

Note: At most 2*(N1+N2-1) comparisons, where N1 = std::distance(first1, last1) and N2 = std::distance(first2, last2).

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *includes* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.

- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *includes* algorithm returns a *bool*. The *includes* algorithm returns true every element from the sorted range [*first2*, *last2*) is found within the sorted range [*first1*, *last1*). Also returns true if [*first2*, *last2*) is empty.

hpx::inclusive_scan

Defined in header hpx/algorithm.hpp[572].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **OutIter**>
OutIter **inclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*, here std::plus<>().

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

[572] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

---

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *inclusive_scan* algorithm returns *OutIter*. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **inclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, *first, …, *(first + (i - result))). Executed according to the policy.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*, here std::plus<>().

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *inclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **OutIter**, typename **Op**>
OutIter **inclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest, *Op* &&op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Op** – The type of the binary function object used for the reduction operation.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *inclusive_scan* algorithm returns *OutIter*. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **inclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, ..., *(first + (i - result))). Executed according to the policy.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, ..., aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, ..., aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Op** – The type of the binary function object used for the reduction operation.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *inclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **OutIter**, typename **Op**, typename **T**>
OutIter **inclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest, *Op* &&op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Op** – The type of the binary function object used for the reduction operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **init** – The initial value for the generalized sum.

Returns The *inclusive_scan* algorithm returns *OutIter*. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **inclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))). Executed according to the policy.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Op** – The type of the binary function object used for the reduction operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **init** – The initial value for the generalized sum.

Returns The *inclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::is_heap, hpx::is_heap_until

Defined in header hpx/algorithm.hpp[573].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_heap**(*ExPolicy* &&policy, *RandIter* first, *RandIter* last, *Comp* &&comp = *Comp*())

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()). Executed according to the policy.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

[573] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Returns The *is_heap* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

template<typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
bool **is_heap**(*RandIter* first, *RandIter* last, *Comp* &&comp = *Comp*())

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **RandIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

Returns The *is_heap* a *bool*. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

template<typename **ExPolicy**, typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *RandIter*> **is_heap_until**(*ExPolicy* &&policy, *RandIter* first, *RandIter* last, *Comp* &&comp = *Comp*())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()). Executed according to the policy.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

Returns The *is_heap_until* algorithm returns a *hpx::future<RandIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandIter* otherwise. The *is_heap_until* algorithm returns the upper bound of the largest range beginning at first which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap.

template<typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
RandIter **is_heap_until**(*RandIter* first, *RandIter* last, *Comp* &&comp = *Comp*())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **RandIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

Returns The *is_heap_until* algorithm returns a *RandIter*. The *is_heap_until* algorithm returns the upper bound of the largest range beginning at first which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap.

hpx::is_partitioned

Defined in header hpx/algorithm.hpp[574].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **Pred**>
bool **is_partitioned**(*FwdIter* first, *FwdIter* last, *Pred* &&pred)

Determines if the range [first, last) is partitioned.

Note: Complexity: at most (N) predicate evaluations where N = distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *is_partitioned* algorithm returns *bool*. The *is_partitioned* algorithm returns true if each element in the sequence for which pred returns true precedes those for which pred returns false. Otherwise is_partitioned returns false. If the range [first, last) contains less than two elements, the function is always true.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred**>

[574] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_partitioned**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Pred* &&pred)

Determines if the range [first, last) is partitioned. Executed according to the policy.

The predicate operations in the parallel *is_partitioned* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_partitioned* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N) predicate evaluations where N = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). *Pred* must be *CopyConstructible* when using a parallel policy.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *is_partitioned* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_partitioned* algorithm returns true if each element in the sequence for which pred returns true precedes those for which pred returns false. Otherwise is_partitioned returns false. If the range [first, last) contains less than two elements, the function is always true.

hpx::is_sorted, hpx::is_sorted_until

Defined in header hpx/algorithm.hpp[575].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **Pred** = *hpx*::*parallel*::detail::less>
bool **is_sorted**(*FwdIter* first, *FwdIter* last, *Pred* &&pred = *Pred*())

Determines if the range [first, last) is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm executes in sequential order in the calling thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use.

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *is_sorted* algorithm returns a *bool*. The *is_sorted* algorithm returns true if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = *hpx*::*parallel*::detail::less>

[575] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

hpx::parallel::util::detail::algorithm_result_t<ExPolicy, bool> **is_sorted**(ExPolicy &&policy, FwdIter first, FwdIter last, Pred &&pred = Pred())

Determines if the range [first, last) is sorted. Uses pred to compare elements. Executed according to the policy.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

  ```
  bool pred(const Type &a, const Type &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *is_sorted* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_sorted* algorithm returns a bool if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

template<typename **FwdIter**, typename **Pred** = hpx::parallel::detail::less>

FwdIter **is_sorted_until**(*FwdIter* first, *FwdIter* last, *Pred* &&pred = *Pred*())

Returns the first element in the range [first, last) that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use.

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *is_sorted_until* algorithm returns a *FwdIter*. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, last is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **is_sorted_until**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Pred* &&pred = *Pred*())

Returns the first element in the range [first, last) that is not sorted. Uses a predicate to compare elements or the less than operator. Executed according to the policy.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted_until* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *is_sorted_until* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, last is returned.

hpx::lexicographical_compare

Defined in header hpx/algorithm.hpp[576].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[576] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **InIter1**, typename **InIter2**, typename **Pred** = *hpx*::*parallel*::detail::less>
bool **lexicographical_compare**(*InIter1* first1, *InIter1* last1, *InIter2* first2, *InIter2* last2, *Pred* &&pred)

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: At most 2 * min(N1, N2) applications of the comparison operation, where N1 = std::distance(first1, last) and N2 = std::distance(first2, last2).

Note: Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Template Parameters

- **InIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **pred** – Refers to the comparison function that the first and second ranges will be applied to

Returns The *lexicographically_compare* algorithm returns a returns *bool* if the execution policy object is not passed in. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **lexicographical_compare**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&pred)

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2 * min(N1, N2) applications of the comparison operation, where N1 = std::distance(first1, last) and N2 = std::distance(first2, last2).

Note: Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **pred** – Refers to the comparison function that the first and second ranges will be applied to

Returns The *lexicographically_compare* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

hpx::make_heap

Defined in header hpx/algorithm.hpp[577].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **RndIter**, typename **Comp**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **make_heap**(*ExPolicy* &&policy, *RndIter* first, *RndIter* last, *Comp* &&comp)

Constructs a *max heap* in the range [first, last). Executed according to the policy.

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

[577] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RndIter** – The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.
- **Comp** – Comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

  ```
  bool cmp(const Type1 &a, const Type2 &b);
  ```

  While the signature does not need to have const &, the function must not modify the objects passed to it and must be able to accept all values of type (possibly *const*) *Type1* and *Type2* regardless of value category (thus, *Type1* & is not allowed, nor is *Type1* unless for *Type1* a move is equivalent to a copy. The types *Type1* and *Type2* must be such that an object of type *RandomIt* can be dereferenced and then implicitly converted to both of them.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **comp** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

  ```
  bool comp(const Type &a, const Type &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *RndIter* can be dereferenced and then implicitly converted to Type.

Returns The *make_heap* algorithm returns a *hpx::future<void>* if the execution policy is of type *task_execution_policy* and returns *void* otherwise.

template<typename **ExPolicy**, typename **RndIter**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **make_heap**(*ExPolicy* &&policy, *RndIter* first, *RndIter* last)

Constructs a *max heap* in the range [first, last). Uses the operator < for comparisons. Executed according to the policy.

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RndIter** – The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.

Returns The *make_heap* algorithm returns a *hpx::future<void>* if the execution policy is of type *task_execution_policy* and returns *void* otherwise.

template<typename **RndIter**, typename **Comp**>
void **make_heap**(*RndIter* first, *RndIter* last, *Comp* &&comp)

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **RndIter** – The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.
- **Comp** – Comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

  While the signature does not need to have const &, the function must not modify the objects passed to it and must be able to accept all values of type (possibly *const*) *Type1* and *Type2* regardless of value category (thus, *Type1* & is not allowed, nor is *Type1* unless for *Type1* a move is equivalent to a copy. The types *Type1* and *Type2* must be such that an object of type *RandomIt* can be dereferenced and then implicitly converted to both of them.

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **comp** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

```
bool comp(const Type &a, const Type &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *RndIter* can be dereferenced and then implicitly converted to Type.

Returns The *make_heap* algorithm returns a *void*.

template<typename **RndIter**>
void **make_heap**(*RndIter* first, *RndIter* last)

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters **RndIter** – The type of the source iterators used for algorithm. This iterator must meet the requirements for a random access iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.

Returns The *make_heap* algorithm returns a *void*.

hpx::merge, hpx::inplace_merge

Defined in header hpx/algorithm.hpp[578].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **RandIter1**, typename **RandIter2**, typename **RandIter3**, typename **Comp** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *RandIter3*> **merge**(*ExPolicy* &&policy, *RandIter1* first1, *RandIter1* last1, *RandIter2* first2, *RandIter2* last2, *RandIter3* dest, *Comp* &&comp = *Comp*())

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges. Executed according to the policy.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

[578] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs O(std::distance(first1, last1) + std::distance(first2, last2)) applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandIter1** – The type of the source iterators used (deduced) representing the first sorted range. This iterator type must meet the requirements of an random access iterator.
- **RandIter2** – The type of the source iterators used (deduced) representing the second sorted range. This iterator type must meet the requirements of an random access iterator.
- **RandIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first range of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first range of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second range of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second range of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter1* and *RandIter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

Returns The *merge* algorithm returns a *hpx::future<RandIter3>* > if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandIter3* otherwise. The *merge* algorithm returns the destination iterator to the end of the *dest* range.

template<typename **RandIter1**, typename **RandIter2**, typename **RandIter3**, typename **Comp** = *hpx*::*parallel*::detail::less>

RandIter3 **merge**(*RandIter1* first1, *RandIter1* last1, *RandIter2* first2, *RandIter2* last2, *RandIter3* dest, *Comp* &&comp = *Comp*())

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

Note: Complexity: Performs O(std::distance(first1, last1) + std::distance(first2, last2)) applications of the comparison *comp* and the each projection.

Template Parameters

- **RandIter1** – The type of the source iterators used (deduced) representing the first sorted range. This iterator type must meet the requirements of an random access iterator.
- **RandIter2** – The type of the source iterators used (deduced) representing the second sorted range. This iterator type must meet the requirements of an random access iterator.
- **RandIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **first1** – Refers to the beginning of the first range of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first range of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second range of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second range of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter1* and *RandIter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

Returns The *merge* algorithm returns a *RandIter3*. The *merge* algorithm returns the destination iterator to the end of the *dest* range.

template<typename **ExPolicy**, typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **inplace_merge**(*ExPolicy* &&policy, *RandIter* first, *RandIter* middle, *RandIter* last, *Comp* &&comp = *Comp*())

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. Executed according to the policy.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

Returns The *inplace_merge* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns void otherwise. The *inplace_merge* algorithm returns the source iterator *last*.

template<typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
void **inplace_merge**(*RandIter* first, *RandIter* middle, *RandIter* last, *Comp* &&comp = *Comp*())

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **RandIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

Returns The *inplace_merge* algorithm returns a *void*. The *inplace_merge* algorithm returns the source iterator *last*.

hpx::min_element, hpx::max_element, hpx::minmax_element

Defined in header hpx/algorithm.hpp[579].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **F** = *hpx*::*parallel*::detail::less>
FwdIter **min_element**(*FwdIter* first, *FwdIter* last, *F* &&f)

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

[579] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

Returns The *min_element* algorithm returns *FwdIter*. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **ExPolicy**, typename **FwdIter**, typename **F** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **min_element**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Finds the smallest element in the range [first, last) using the given comparison function *f*. Executed according to the policy.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *min_element* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

Returns The *min_element* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **FwdIter**, typename **F** = *hpx*::*parallel*::detail::less>
FwdIter **max_element**(*FwdIter* first, *FwdIter* last, *F* &&f)

Finds the largest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

Returns The *max_element* algorithm returns *FwdIter*. The *max_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **ExPolicy**, typename **FwdIter**, typename **F** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **max_element**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Removes all elements satisfying specific criteria from the range Finds the largest element in the range [first, last) using the given comparison function *f*. Executed according to the policy.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *max_element* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

Returns The *max_element* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *max_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **FwdIter**, typename **F** = *hpx*::*parallel*::detail::less>
minmax_element_result<*FwdIter*> **minmax_element**(*FwdIter* first, *FwdIter* last, *F* &&f)

Finds the largest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *minmax_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: At most *max*(floor(3/2*(N-1)), 0) applications of the predicate, where N = std::distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. This argument is optional and defaults to std::less. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

Returns The *minmax_element* algorithm returns a *minmax_element_result<FwdIter>* The *minmax_element* algorithm returns a pair consisting of an iterator to the smallest element as the min element and an iterator to the largest element as the max element. Returns *minmax_element_result<FwdIter>{first*,first} if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **F** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, minmax_element_result<*FwdIter*>> **minmax_element**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *F* &&f)

Finds the largest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *max*(floor(3/2*(N-1)), 0) applications of the predicate, where N = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *minmax_element* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. This argument is optional and defaults to std::less. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

Returns The *minmax_element* algorithm returns a *hpx::future<minmax_element_result<FwdIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *minmax_element_result<FwdIter>* otherwise. The *minmax_element* algorithm returns a pair consisting of an iterator to the smallest element as the min element and an iterator to the largest element as the max element. Returns *std::make_pair(first,first)* if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

hpx::mismatch

Defined in header hpx/algorithm.hpp[580].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *std*::pair<*FwdIter1*, *FwdIter2*>> **mismatch**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1). Executed according to the policy.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *op* or *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of the predicate *op* or *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

[580] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *mismatch* algorithm returns a *hpx::future<std::pair<FwdIter1,FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1,FwdIter2>* otherwise. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *std*::pair<*FwdIter1*, *FwdIter2*>> **mismatch**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1). Executed according to the policy.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.

Returns The *mismatch* algorithm returns a *hpx::future<std::pair<FwdIter1,FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1,FwdIter2>* otherwise. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *std*::pair<*FwdIter1*, *FwdIter2*>> **mismatch**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *Pred* &&op)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1). Executed according to the policy.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last1 - first1 applications of the predicate *op* or *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of the predicate *op* or *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.

- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *mismatch* algorithm returns a *hpx::future<std::pair<FwdIter1,FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1,FwdIter2>* otherwise. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *std*::pair<*FwdIter1*, *FwdIter2*>> **mismatch**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1). Executed according to the policy.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last1 - first1 applications of *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.

Returns The *mismatch* algorithm returns a *hpx::future<std::pair<FwdIter1,FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1,FwdIter2>* otherwise. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>
std::pair<*FwdIter1*, *FwdIter2*> **mismatch**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *Pred* &&op)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1).

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *op* or *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of the predicate *op* or *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *mismatch* algorithm returns a *std::pair<FwdIter1,FwdIter2>*. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

template<typename **FwdIter1**, typename **FwdIter2**>
std::pair<*FwdIter1*, *FwdIter2*> **mismatch**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1).

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.

Returns The *mismatch* algorithm returns a *std::pair<FwdIter1,FwdIter2>*. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

template<typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>
std::pair<*FwdIter1*, *FwdIter2*> **mismatch**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *Pred* &&op)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1).

Note: Complexity: At most last1 - first1 applications of the predicate *op* or *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of the predicate *op* or *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *mismatch* algorithm returns a *std::pair<FwdIter1,FwdIter2>*. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

template<typename **FwdIter1**, typename **FwdIter2**>
std::pair<*FwdIter1*, *FwdIter2*> **mismatch**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2)

Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided, it denotes first2 + (last1 - first1).

Note: Complexity: At most last1 - first1 applications of *operator==*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of *operator==* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatches *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.

Returns The *mismatch* algorithm returns a *std::pair<FwdIter1,FwdIter2>*. If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

hpx::move

Defined in header hpx/algorithm.hpp[581].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **move**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Moves the elements in the range [first, last), to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move. Executed according to the policy.

The move assignments in the parallel *move* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The move assignments in the parallel *move* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* move assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the move assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns a *hpx::future<FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *move* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

[581] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

template<typename **FwdIter1**, typename **FwdIter2**>
FwdIter2 **move**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Moves the elements in the range [first, last), to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

Note: Complexity: Performs exactly *last* - *first* move assignments.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns a *FwdIter2*. The *move* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

hpx::nth_element

Defined in header hpx/algorithm.hpp[582].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **RandomIt**, typename **Pred** = *hpx*::*parallel*::detail::less>
void **nth_element**(*RandomIt* first, *RandomIt* nth, *RandomIt* last, *Pred* &&pred = *Pred*())

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by nth is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new nth element are less than or equal to the elements after the new nth element. Executed according to the policy.

The comparison operations in the parallel *nth_element* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

[582] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Template Parameters

- **RandomIt** – The type of the source begin, nth, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Pred** – Comparison function object which returns true if the first argument is less than the second. This defaults to std::less<>.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **nth** – Refers to the iterator defining the sort partition point
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.

Returns The *nth_element* algorithms returns nothing.

template<typename **ExPolicy**, typename **RandomIt**, typename **Pred** = *hpx*::*parallel*::detail::less>
void **nth_element**(*ExPolicy* &&policy, *RandomIt* first, *RandomIt* nth, *RandomIt* last, *Pred* &&pred = *Pred*())

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by nth is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new nth element are less than or equal to the elements after the new nth element.

The comparison operations in the parallel *nth_element* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *nth_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandomIt** – The type of the source begin, nth, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.

- **Pred** – Comparison function object which returns true if the first argument is less than the second. This defaults to std::less<>.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **nth** – Refers to the iterator defining the sort partition point
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.

Returns The *nth_element* algorithms returns nothing.

hpx::partial_sort

Defined in header hpx/algorithm.hpp[583].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
RandIter **partial_sort**(*RandIter* first, *RandIter* middle, *RandIter* last, *Comp* &&comp = *Comp*())

Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

Note: Complexity: Approximately (last - first) * log(middle - first) comparisons.

Template Parameters

- **RandIter** – The type of the source begin, middle, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

[583] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **middle** – Refers to the middle of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. It defaults to detail::less.

Returns The *partial_sort* algorithm returns a *RandIter* that refers to *last*.

template<typename **ExPolicy**, typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *RandIter*> **partial_sort**(*ExPolicy* &&policy, *RandIter* first, *RandIter* middle, *RandIter* last, *Comp* &&comp = *Comp*())

Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

Note: Complexity: Approximately (last - first) * log(middle - first) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **RandIter** – The type of the source begin, middle, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the middle of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. It defaults to detail::less.

Returns The *partial_sort* algorithm returns a *hpx::future<RandIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandIter* otherwise. The iterator returned refers to *last*.

hpx::partial_sort_copy

Defined in header hpx/algorithm.hpp[584].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
RandIter **partial_sort_copy**(*InIter* first, *InIter* last, *RandIter* d_first, *RandIter* d_last, *Comp* &&comp = *Comp*())

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [d_first, d_last). At most d_last - d_first of the elements are placed sorted to the range [d_first, d_first + n) where n is the number of elements to sort (n = min(last - first, d_last - d_first)).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(min(D,N))), where N = std::distance(first, last) and D = std::distance(d_first, d_last) comparisons.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **RandIter** – The type of the destination iterators used(deduced) This iterator type must meet the requirements of an random iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **d_first** – Refers to the beginning of the destination range.
- **d_last** – Refers to the end of the destination range.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to detail::less.

[584] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Returns The *partial_sort_copy* algorithm returns a *RandomIt*. The algorithm returns an iterator to the element defining the upper boundary of the sorted range i.e. d_first + min(last - first, d_last - d_first)

template<typename **ExPolicy**, typename **FwdIter**, typename **RandIter**, typename **Comp** = *hpx*::*parallel*::detail::less>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *RandIter*> **partial_sort_copy**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *RandIter* d_first, *RandIter* d_last, *Comp* &&comp = *Comp*())

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [d_first, d_last). At most d_last - d_first of the elements are placed sorted to the range [d_first, d_first + n) where n is the number of elements to sort (n = min(last - first, d_last - d_first)). Executed according to the policy.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(min(D,N))), where N = std::distance(first, last) and D = std::distance(d_first, d_last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **RandIter** – The type of the destination iterators used(deduced) This iterator type must meet the requirements of an random iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **d_first** – Refers to the beginning of the destination range.
- **d_last** – Refers to the end of the destination range.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to detail::less.

Returns The *partial_sort_copy* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator to the element defining the upper boundary of the sorted range i.e. d_first + min(last - first, d_last - d_first)

hpx::partition, hpx::stable_partition, hpx::partition_copy

Defined in header hpx/algorithm.hpp[585].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **Pred**, typename **Proj** = *hpx*::identity>
FwdIter **partition**(*FwdIter* first, *FwdIter* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: At most 2 * (last - first) swaps. Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

[585] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition* algorithm returns returns *FwdIter*. The *partition* algorithm returns the iterator to the first element of the second group.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **partition**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2 * (last - first) swaps. Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *parallel_task_policy* and returns *FwdIter* otherwise. The *partition* algorithm returns the iterator to the first element of the second group.

template<typename **BidirIter**, typename **F**, typename **Proj** = *hpx*::identity>
BidirIter **stable_partition**(*BidirIter* first, *BidirIter* last, *F* &&f, *Proj* &&proj = *Proj*())

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) INVOKE(f, INVOKE (proj, *j)) != false, and for every iterator k in the range [i, last), INVOKE(f, INVOKE (proj, *k)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked without an execution policy object executes in sequential order in the calling thread.

Note: Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory. Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a bidirectional iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range [first, i), f(*j) != false INVOKE(f, INVOKE(proj, *j)) != false, and for every iterator k in the range [i, last), f(*k) == false INVOKE(f, INVOKE (proj, *k)) == false. The relative order of the elements in both groups is preserved.

template<typename **ExPolicy**, typename **BidirIter**, typename **F**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *BidirIter*> **stable_partition**(*ExPolicy* &&policy, *BidirIter* first, *BidirIter* last, *F* &&f, *Proj* &&proj = *Proj*())

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) INVOKE(f, INVOKE (proj, *j)) != false, and for every iterator k in the range [i, last), INVOKE(f, INVOKE (proj, *k)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory. Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a bidirectional iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  bool fun(const Type &a);
  ```

  The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range [first, i), f(*j) != false INVOKE(f, INVOKE(proj, *j)) != false, and for every iterator k in the range [i, last), f(*k) == false INVOKE(f, INVOKE (proj, *k)) == false. The relative order of the elements in both groups is preserved. If the execution policy is of type *parallel_task_policy* the algorithm returns a *future*<> referring to this iterator.

template<typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred**, typename **Proj** = *hpx*::identity>
std::pair<*FwdIter2*, *FwdIter3*> **partition_copy**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* dest_true, *FwdIter3* dest_false, *Pred* &&pred, *Proj* &&proj = *Proj*())

Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred*.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest_true** – Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- **dest_false** – Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.

- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition_copy* algorithm returns *std::pair<OutIter1, OutIter2>*. The *partition_copy* algorithm returns the pair of the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *std*::pair<*FwdIter2*, *FwdIter3*>> **partition_copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest_true, *FwdIter3* dest_false, *Pred* &&pred, *Proj* &&proj = *Proj*())

Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred*, are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest_true** – Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- **dest_false** – Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition_copy* algorithm returns a *hpx::future<std::pair<OutIter1, OutIter2>>* if the execution policy is of type *parallel_task_policy* and returns *std::pair<OutIter1, OutIter2>* otherwise. The *partition_copy* algorithm returns the pair of the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

hpx::reduce

Defined in header hpx/algorithm.hpp[586].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **F**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **reduce**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* init, *F* &&f)

Returns GENERALIZED_SUM(f, init, *first, …, *(first + (last - first) - 1)). Executed according to the policy.

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicate *f*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *reduce* requires *F* to meet the requirements of *CopyConstructible*.

[586] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&. The types *Type1* *Ret* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to any of those types.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **FwdIter**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
util::detail::algorithm_result_t<*ExPolicy*, *T*> **reduce**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* init)

Returns GENERALIZED_SUM(+, init, *first, …, *(first + (last - first) - 1)). Executed according to the policy.

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **FwdIter**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<*FwdIter*>::value_type>::type **reduce**(*ExPoli* &&policy, *FwdIte* first, *FwdIte* last)

Returns GENERALIZED_SUM(+, T(), *first, …, *(first + (last - first) - 1)). Executed according to the policy.

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the value_type of the used *FwdIter*.

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns T otherwise (where T is the value_type of *FwdIter*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **FwdIter**, typename **F**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
T **reduce**(*FwdIter* first, *FwdIter* last, *T* init, *F* &&f)

Returns GENERALIZED_SUM(f, init, *first, …, *(first + (last - first) - 1)). Executed according to the policy.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicate *f*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **FwdIter** – The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *reduce* requires *F* to meet the requirements of *CopyConstructible*.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&. The types *Type1* *Ret* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to any of those types.

Returns The *reduce* algorithm returns *T*. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

template<typename **FwdIter**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
T **reduce**(*FwdIter* first, *FwdIter* last, *T* init)

Returns GENERALIZED_SUM(+, init, *first, …, *(first + (last - first) - 1)). Executed according to the policy.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **FwdIter** – The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns a *T*. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **FwdIter**>
std::iterator_traits<*FwdIter*>::value_type **reduce**(*FwdIter* first, *FwdIter* last)

Returns GENERALIZED_SUM(+, T(), *first, …, *(first + (last - first) - 1)). Executed according to the policy.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the value_type of the used *FwdIter*.

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters **FwdIter** – The type of the source begin and end iterators used (deduced). This iterator type must meet the requirements of an input iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns *T* (where T is the value_type of *FwdIter*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

hpx::reduce_by_key

Defined in header hpx/algorithm.hpp[587].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **experimental**

Top-level namespace.

Functions

template<typename **ExPolicy**, typename **RanIter**, typename **RanIter2**, typename **FwdIter1**, typename **FwdIter2**, typename **Compare** = *std*::equal_to<typename *std*::iterator_traits<*RanIter*>::value_type>, typename **Func** = *std*::plus<typename *std*::iterator_traits<*RanIter2*>::value_type>>
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<*FwdIter1*, *FwdIter2*>>::type **reduce_by_key**(*ExPolicy* &&policy, *RanIter* key_first, *RanIter* key_last, *RanIter2* values_first, *FwdIter1* keys_output, *FwdIter2* values_output, *Compare* &&comp = *Compare*(), *Func* &&func = *Func*())

Reduce by Key performs an inclusive scan reduction operation on elements supplied in key/value pairs. The algorithm produces a single output value for each set of equal consecutive keys in [key_first, key_last). the value being the GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))). for the run of consecutive matching keys. The number of keys supplied must match the number of values.

[587] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **RanIter** – The type of the key iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **RanIter2** – The type of the value iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **FwdIter1** – The type of the iterator representing the destination key range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the iterator representing the destination value range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Compare** – The type of the optional function/function object to use to compare keys (deduced). Assumed to be std::equal_to otherwise.
- **Func** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *reduce_by_key* requires *Func* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **key_first** – Refers to the beginning of the sequence of key elements the algorithm will be applied to.
- **key_last** – Refers to the end of the sequence of key elements the algorithm will be applied to.
- **values_first** – Refers to the beginning of the sequence of value elements the algorithm will be applied to.
- **keys_output** – Refers to the start output location for the keys produced by the algorithm.
- **values_output** – Refers to the start output location for the values produced by the algorithm.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **func** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to any of those types.

Returns The *reduce_by_key* algorithm returns a *hpx::future<pair<Iter1,Iter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *pair<Iter1,Iter2>* otherwise.

hpx::remove, hpx::remove_if

Defined in header hpx/algorithm.hpp[588].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
FwdIter **remove**(*FwdIter* first, *FwdIter* last, *T* const &value)

Removes all elements satisfying specific criteria from the range [first, last) and returns a past-the-end iterator for the new end of the range. This version removes all elements that are equal to *value*.

The assignments in the parallel *remove* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the operator==().

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – Specifies the value of elements to remove.

Returns The *remove* algorithm returns a *FwdIter*. The *remove* algorithm returns the iterator to the new end of the range.

template<typename **ExPolicy**, typename **FwdIter**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **remove**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* const &value)

Removes all elements satisfying specific criteria from the range [first, last) and returns a past-the-end iterator for the new end of the range. This version removes all elements that are equal to *value*. Executed according to the policy.

[588] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the operator==().

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`FwdIter`** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **`T`** – The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **`last`** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **`value`** – Specifies the value of elements to remove.

Returns The *remove* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove* algorithm returns the iterator to the new end of the range.

template<typename **FwdIter**, typename **Pred**>
FwdIter **remove_if**(*FwdIter* first, *FwdIter* last, *Pred* &&pred)

Removes all elements satisfying specific criteria from the range [first, last) and returns a past-the-end iterator for the new end of the range. This version removes all elements for which predicate *pred* returns true.

The assignments in the parallel *remove_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred*.

Template Parameters

- **`FwdIter`** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **`Pred`** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *remove_if* algorithm returns a *FwdIter*. The *remove_if* algorithm returns the iterator to the new end of the range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **remove_if**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Pred* &&pred)

Removes all elements satisfying specific criteria from the range [first, last) and returns a past-the-end iterator for the new end of the range. This version removes all elements for which predicate *pred* returns true. Executed according to the policy.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

Returns The *remove_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove_if* algorithm returns the iterator to the new end of the range.

hpx::remove_copy, hpx::remove_copy_if

Defined in header hpx/algorithm.hpp[589].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **OutIter**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type>
OutIter **remove_copy**(*InIter* first, *InIter* last, *OutIter* dest, *T* const &value)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the comparison operator returns false when compare to value. The order of the elements that are not removed is preserved.

Effects: Copies all the elements referred to by the iterator it in the range [first,last) for which the following corresponding conditions do not hold: *it == value

The assignments in the parallel *remove_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last - first* assignments, exactly *last - first* applications of the predicate *pred*, here comparison operator.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T** – The type that the result of dereferencing FwdIter1 is compared to.

Parameters

[589] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **value** – Value to be removed.

Returns The *remove_copy* algorithm returns an *OutIter*. The *remove_copy* algorithm returns the iterator to the element past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T** = typename *std*::iterator_traits<InIter>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **remove_copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *T* const &value)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the comparison operator returns false when compare to value. The order of the elements that are not removed is preserved. Executed according to the policy.

Effects: Copies all the elements referred to by the iterator it in the range [first,last) for which the following corresponding conditions do not hold: *it == value

The assignments in the parallel *remove_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred*, here comparison operator.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type that the result of dereferencing FwdIter1 is compared to.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

- **dest** – Refers to the beginning of the destination range.
- **value** – Value to be removed.

Returns The *remove_copy* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *remove_copy* algorithm returns the iterator to the element past the last element copied.

template<typename **InIter**, typename **OutIter**, typename **Pred**>
OutIter **remove_copy_if**(*InIter* first, *InIter* last, *OutIter* dest, *Pred* &&pred)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the predicate *pred* returns false. The order of the elements that are not removed is preserved.

Effects: Copies all the elements referred to by the iterator it in the range [first,last) for which the following corresponding conditions do not hold: INVOKE(pred, *it) != false.

The assignments in the parallel *remove_copy_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred*.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of the function/function object to use (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements to be removed. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

Returns The *remove_copy_if* algorithm returns an *OutIter* The *remove_copy_if* algorithm returns the iterator to the element past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred**>

hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter2> **remove_copy_if**(ExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest, Pred &&pred)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*. Copies only the elements for which the predicate *pred* returns false. The order of the elements that are not removed is preserved. Executed according to the policy.

Effects: Copies all the elements referred to by the iterator it in the range [first,last) for which the following corresponding conditions do not hold: INVOKE(pred, *it) != false.

The assignments in the parallel *remove_copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove_copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_copy_if* requires *Pred* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements to be removed. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.

Returns The *remove_copy_if* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *remove_copy_if* algorithm returns the iterator to the element past the last element copied.

hpx::replace, hpx::replace_if, hpx::replace_copy, hpx::replace_copy_if

Defined in header hpx/algorithm.hpp[590].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type>
void **replace**(*InIter* first, *InIter* last, *T* const &old_value, *T* const &new_value)

Replaces all elements satisfying specific criteria with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first, last) with new_value, when the following corresponding conditions hold: *it == old_value

The assignments in the parallel *replace* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the old and new values to replace (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace* algorithm returns a *void*.

template<typename **ExPolicy**, typename **FwdIter**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>

[590] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, void> **replace**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* const &old_value, *T* const &new_value)

Replaces all elements satisfying specific criteria with *new_value* in the range [first, last). Executed according to the policy.

Effects: Substitutes elements referred by the iterator it in the range [first, last) with new_value, when the following corresponding conditions hold: *it == old_value

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **T** – The type of the old and new values to replace (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **Iter**, typename **Pred**, typename **T** = typename *std*::iterator_traits<*Iter*>::value_type>
void **replace_if**(*Iter* first, *Iter* last, *Pred* &&pred, *T* const &new_value)

Replaces all elements satisfying specific criteria (for which predicate *pred* returns true) with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first, last) with new_value, when the following corresponding conditions hold: INVOKE(f, *it) != false

The assignments in the parallel *replace_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace_if* algorithm returns *void*.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, void> **replace_if**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Pred* &&pred, *T* const &new_value)

Replaces all elements satisfying specific criteria (for which predicate *f* returns true) with *new_value* in the range [first, last). Executed according to the policy.

Effects: Substitutes elements referred by the iterator it in the range [first, last) with new_value, when the following corresponding conditions hold: INVOKE(f, *it) != false

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace_if* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **InIter**, typename **OutIter**, typename **T** = typename *std*::iterator_traits<*OutIter*>::value_type>
OutIter **replace_copy**(*InIter* first, *InIter* last, *OutIter* dest, *T* const &old_value, *T* const &new_value)

Copies the all elements from the range [first, last) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (last - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: *(first + (i - result)) == old_value

The assignments in the parallel *replace_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.

- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T** – The type of the old and new values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace_copy* algorithm returns an *OutIter* The *replace_copy* algorithm returns the Iterator to the element past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T** = typename *std*::iterator_traits<*FwdIter2*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **replace_copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *T* const &old_value, *T* const &new_value)

Copies the all elements from the range [first, last) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*. Executed according to the policy.

Effects: Assigns to every iterator it in the range [result, result + (last - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: *(first + (i - result)) == old_value

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the old and new values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace_copy* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *replace_copy* algorithm returns the Iterator to the element past the last element copied.

template<typename **InIter**, typename **OutIter**, typename **Pred**, typename **T** = typename *std*::iterator_traits<*OutIter*>::value_type>
OutIter **replace_copy_if**(*InIter* first, *InIter* last, *OutIter* dest, *Pred* &&pred, *T* const &new_value)

Copies the all elements from the range [first, last) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (last - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(f, *(first + (i - result))) != false

The assignments in the parallel *replace_copy_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*.

- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace_copy_if* algorithm returns an *OutIter*. The *replace_copy_if* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred**, typename **T** = typename *std*::iterator_traits<*FwdIter2*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **replace_copy_if**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Pred* &&pred, *T* const &new_value)

Copies the all elements from the range [first, last) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (last - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(f, *(first + (i - result))) != false

The assignments in the parallel *replace_copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *replace_copy_if* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.

Returns The *replace_copy_if* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *replace_copy_if* algorithm returns the iterator to the element in the destination range, one past the last element copied.

hpx::reverse, hpx::reverse_copy

Defined in header hpx/algorithm.hpp[591].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **BidirIter**>
void **reverse**(*BidirIter* first, *BidirIter* last)

Reverses the order of the elements in the range [first, last). Behaves as if applying *std::iter_swap* to every pair of iterators first+i, (*last-i*) - 1 for each non-negative i < (last-first)/2.

The assignments in the parallel *reverse* algorithm execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters BidirIter – The type of the source iterators used (deduced). This iterator type must meet the requirements of a bidirectional iterator.

Parameters

[591] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reverse* algorithm returns *void*.

template<typename **ExPolicy**, typename **BidirIter**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, void> **reverse**(*ExPolicy* &&policy, *BidirIter* first, *BidirIter* last)

Reverses the order of the elements in the range [first, last). Behaves as if applying *std::iter_swap* to every pair of iterators first+i, (last-i) - 1 for each non-negative i < (last-first)/2. Executed according to the policy.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a bidirectional iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reverse* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **BidirIter**, typename **OutIter**>
OutIter **reverse_copy**(*BidirIter* first, *BidirIter* last, *OutIter* dest)

Copies the elements from the range [first, last) to another range beginning at dest in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(dest + (last - first) - 1 - i) = *(first + i) once for each non-negative i < (last - first) If the source and destination ranges (that is, [first, last) and [dest, dest+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a bidirectional iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the begin of the destination range.

Returns The *reverse_copy* algorithm returns an *OutIter*. The *reverse_copy* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **BidirIter**, typename **FwdIter**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **reverse_copy**(*ExPolicy* &&policy, *BidirIter* first, *BidirIter* last, *FwdIter* dest)

Copies the elements from the range [first, last) to another range beginning at dest in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(dest + (last - first) - 1 - i) = *(first + i) once for each non-negative i < (last - first) If the source and destination ranges (that is, [first, last) and [dest, dest+(last-first)) respectively) overlap, the behavior is undefined. Executed according to the policy.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a bidirectional iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

- **dest** – Refers to the begin of the destination range.

Returns The *reverse_copy* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *reverse_copy* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::rotate, hpx::rotate_copy

Defined in header hpx/algorithm.hpp[592].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**>
FwdIter **rotate**(*FwdIter* first, *FwdIter* new_first, *FwdIter* last)

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element new_first becomes the first element of the new range and new_first - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Template Parameters FwdIter – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **new_first** – Refers to the element that should appear at the beginning of the rotated range.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *rotate* algorithm returns a FwdIter. The *rotate* algorithm returns the iterator to the new location of the element pointed by first,equal to first + (last - new_first).

template<typename **ExPolicy**, typename **FwdIter**>

[592] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **rotate**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* new_first, *FwdIter* last)

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element new_first becomes the first element of the new range and new_first - 1 becomes the last element. Executed according to the policy.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **new_first** – Refers to the element that should appear at the beginning of the rotated range.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *rotate* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *rotate* algorithm returns the iterator equal to first + (last - new_first).

template<typename **FwdIter**, typename **OutIter**>
OutIter **rotate_copy**(*FwdIter* first, *FwdIter* new_first, *FwdIter* last, *OutIter* dest_first)

Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *new_first* becomes the first element of the new range and *new_first* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- `FwdIter` – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `OutIter` – The type of the source iterators used (deduced). This iterator type must meet the requirements of a output iterator.

Parameters

- `first` – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `new_first` – Refers to the element that should appear at the beginning of the rotated range.
- `last` – Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest_first` – Refers to the begin of the destination range.

Returns The *rotate_copy* algorithm returns a output iterator, The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **rotate_copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* new_first, *FwdIter1* last, *FwdIter2* dest_first)

Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *new_first* becomes the first element of the new range and *new_first* - 1 becomes the last element. Executed according to the policy.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

The assignments in the parallel *rotate_copy* algorithm execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- `ExPolicy` – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1` – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `FwdIter2` – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- `policy` – The execution policy to use for the scheduling of the iterations.
- `first` – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `new_first` – Refers to the element that should appear at the beginning of the rotated range.
- `last` – Refers to the end of the sequence of elements the algorithm will be applied to.

- **dest_first** – Refers to the begin of the destination range.

Returns The *rotate_copy* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *parallel_task_policy* and returns FwdIter2 otherwise. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

hpx::search, hpx::search_n

Defined in header hpx/algorithm.hpp[593].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **FwdIter2**, typename **Pred** = *parallel*::detail::equal_to>
FwdIter **search**(*FwdIter* first, *FwdIter* last, *FwdIter2* s_first, *FwdIter2* s_last, *Pred* &&op = *Pred*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *search* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to<>*

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.

[593] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **FwdIter2**, typename **Pred** = *parallel*::detail::equal_to>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **search**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *FwdIter2* s_first, *FwdIter2* s_last, *Pred* &&op = *Pred*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements. Executed according to the policy.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *search* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to<>*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

template<typename **FwdIter**, typename **FwdIter2**, typename **Pred** = *parallel*::detail::equal_to>
FwdIter **search_n**(*FwdIter* first, *std*::size_t count, *FwdIter2* s_first, *FwdIter2* s_last, *Pred* &&op = *Pred*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search_n* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = count.

Template Parameters

- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *search_n* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to<>*

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **count** – Refers to the range of elements of the first range the algorithm will be applied to.

- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *search_n* algorithm returns *FwdIter*. The *search_n* algorithm returns an iterator to the beginning of the last subsequence [s_first, s_last) in range [first, first+count). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, first+count), *first* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *first* is also returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **FwdIter2**, typename **Pred** = *parallel*::detail::equal_to>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **search_n**(*ExPolicy* &&policy, *FwdIter* first, *std*::size_t count, *FwdIter2* s_first, *FwdIter2* s_last, *Pred* &&op = *Pred*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements. Executed according to the policy.

The comparison operations in the parallel *search_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *search_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = count.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *search_n* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to *std::equal_to*<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **count** – Refers to the range of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

Returns The *search_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search_n* algorithm returns an iterator to the beginning of the last subsequence [s_first, s_last) in range [first, first+count). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, first+count), *first* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *first* is also returned.

hpx::set_difference

Defined in header hpx/algorithm.hpp[594].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter3*> **set_difference**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *pred*. Executed according to the policy.

[594] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_difference* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>
FwdIter3 **set_difference**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *pred*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.

- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_difference* algorithm returns a *FwdIter3*. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::set_intersection

Defined in header hpx/algorithm.hpp[595].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter3*> **set_intersection**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *pred*. Executed according to the policy.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), the first std::min(m, n) elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

[595] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator with sequential execution.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_intersection* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>

FwdIter3 **set_intersection**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *pred*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), the first std::min(m, n) elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator with sequential execution.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_intersection* algorithm returns a *FwdIter3*. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::set_symmetric_difference

Defined in header hpx/algorithm.hpp[596].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter3*>::type **set_symmetric_difference**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in either of the sorted ranges [first1, last1) and [first2, last2), but not in both of them are copied to the range beginning at *dest*. The resulting range is also sorted. This algorithm expects both input ranges to be sorted with the given binary predicate *pred*. Executed according to the policy.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::abs(m-n) times. If m>n, then the last m-n of those elements are copied from [first1,last1), otherwise the last n-m elements are copied from [first2,last2). The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

[596] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_symmetric_difference* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>
FwdIter3 **set_symmetric_difference**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in either of the sorted ranges [first1, last1) and [first2, last2), but not in both of them are copied to the range beginning at *dest*. The resulting range is also sorted. This algorithm expects both input ranges to be sorted with the given binary predicate *pred*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::abs(m-n) times. If m>n, then the last m-n of those elements are copied from [first1,last1), otherwise the last n-m elements are copied from [first2,last2). The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_symmetric_difference* algorithm returns a *FwdIter3*. The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::set_union

Defined in header hpx/algorithm.hpp[597].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter3*> **set_union**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in one or both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *pred*. Executed according to the policy.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), then all *m* elements will be copied from [first1, last1) to dest, preserving order, and then exactly std::max(n-m, 0) elements will be copied from [first2, last2) to dest, also preserving order.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

[597] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.
- **Op** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_union* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_union* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter3* otherwise. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred** = *hpx*::*parallel*::detail::less>
FwdIter3 **set_union**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter2* last2, *FwdIter3* dest, *Pred* &&op = *Pred*())

Constructs a sorted range beginning at dest consisting of all elements present in one or both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *pred*. Executed according to the policy.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), then all *m* elements will be copied from [first1, last1) to dest, preserving order, and then exactly std::max(n-m, 0) elements will be copied from [first2, last2) to dest, also preserving order.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **`FwdIter1`** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **`FwdIter2`** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of a forward iterator.
- **`FwdIter3`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator or output iterator and sequential execution.
- **`Op`** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_union* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- **`first1`** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **`last1`** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **`first2`** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **`last2`** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **`dest`** – Refers to the beginning of the destination range.
- **`op`** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*

Returns The *set_union* algorithm returns a *FwdIter3*. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::shift_left

Defined in header hpx/algorithm.hpp[598].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **Size**>
FwdIter **shift_left**(*FwdIter* first, *FwdIter* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i.

The assignment operations in the parallel *shift_left* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **n** – Refers to the number of positions to shift.

Returns The *shift_left* algorithm returns *FwdIter*. The *shift_left* algorithm returns an iterator to the end of the resulting range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*> **shift_left**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i. Executed according to the policy.

[598] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **n** – Refers to the number of positions to shift.

Returns The *shift_left* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *shift_left* algorithm returns an iterator to the end of the resulting range.

hpx::shift_right

Defined in header hpx/algorithm.hpp[599].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[599] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **FwdIter**, typename **Size**>
FwdIter **shift_right**(*FwdIter* first, *FwdIter* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- n + i.

The assignment operations in the parallel *shift_right* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **n** – Refers to the number of positions to shift.

Returns The *shift_right* algorithm returns *FwdIter*. The *shift_right* algorithm returns an iterator to the end of the resulting range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*> **shift_right**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- n + i. Executed according to the policy.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **n** – Refers to the number of positions to shift.

Returns The *shift_right* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *shift_right* algorithm returns an iterator to the end of the resulting range.

hpx::sort

Defined in header hpx/algorithm.hpp[600].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **RandomIt**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
void **sort**(*RandomIt* first, *RandomIt* last, *Comp* &&comp, *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

[600] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *sort* algorithm returns *void*.

template<typename **ExPolicy**, typename **RandomIt**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **sort**(*ExPolicy* &&policy, *RandomIt* first, *RandomIt* last, *Comp* &&comp, *Proj* &&proj)

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()). Executed according to the policy.

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.

- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *sort* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

hpx::experimental::sort_by_key

Defined in header hpx/algorithm.hpp[601].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **experimental**

Top-level namespace.

Functions

template<typename **ExPolicy**, typename **KeyIter**, typename **ValueIter**, typename **Compare** = detail::less>

[601] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

util::detail::algorithm_result_t<*ExPolicy*, sort_by_key_result<*KeyIter*, *ValueIter*>> **sort_by_key**(*ExPolicy* &&policy, *KeyIter* key_first, *KeyIter* key_last, *ValueIter* value_first, *Compare* &&comp = *Compare*())

Sorts one range of data using keys supplied in another range. The key elements in the range [key_first, key_last) are sorted in ascending order with the corresponding elements in the value range moved to follow the sorted order. The algorithm is not stable, the order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()). Executed according to the policy.

A sequence is sorted with respect to a comparator *comp* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, *(i + n), *i) == false.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **KeyIter** – The type of the key iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **ValueIter** – The type of the value iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Compare** – The type of the function/function object to use (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **key_first** – Refers to the beginning of the sequence of key elements the algorithm will be applied to.
- **key_last** – Refers to the end of the sequence of key elements the algorithm will be applied to.

- **value_first** – Refers to the beginning of the sequence of value elements the algorithm will be applied to, the range of elements must match [key_first, key_last)
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

Returns The *sort_by_key* algorithm returns a *hpx::future<sort_by_key_result<KeyIter,ValueIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *otherwise*. The algorithm returns a pair holding an iterator pointing to the first element after the last element in the input key sequence and an iterator pointing to the first element after the last element in the input value sequence.

hpx::stable_sort

Defined in header hpx/algorithm.hpp[602].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **RandomIt**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
void **stable_sort**(*RandomIt* first, *RandomIt* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *stable_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

602 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *stable_sort* algorithm returns *void*.

template<typename **ExPolicy**, typename **RandomIt**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **stable_sort**(*ExPolicy* &&policy, *RandomIt* first, *RandomIt* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()). Executed according to the policy.

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *stable_sort* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

hpx::starts_with

Defined in header hpx/algorithm.hpp[603].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter1**, typename **InIter2**, typename **Pred** = *hpx*::*parallel*::detail::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **starts_with**(*InIter1* first1, *InIter1* last1, *InIter2* first2, *InIter2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range defined by [first1, last1) matches the prefix of the first range defined by [first2, last2)

The assignments in the parallel *starts_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **InIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the destination iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Pred** – The binary predicate that compares the projected elements. This defaults to *hpx::parallel::detail::equal_to*.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*.
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*.

Parameters

[603] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by *proj1* and *proj2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *pred* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *pred* is invoked.

Returns The *starts_with* algorithm returns *bool*. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

template<typename **ExPolicy**, typename **InIter1**, typename **InIter2**, typename **Pred** = *hpx*::*parallel*::detail::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **starts_with**(*ExPolicy* &&policy, *InIter1* first1, *InIter1* last1, *InIter2* first2, *InIter2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range defined by [first1, last1) matches the prefix of the first range defined by [first2, last2). Executed according to the policy.

The assignments in the parallel *starts_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **InIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the destination iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Pred** – The binary predicate that compares the projected elements. This defaults to *hpx::parallel::detail::equal_to*.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*.
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by *proj1* and *proj2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *pred* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *pred* is invoked.

Returns The *starts_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

hpx::swap_ranges

Defined in header hpx/algorithm.hpp[604].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter1**, typename **FwdIter2**>
FwdIter2 **swap_ranges**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2)

Exchanges elements between range [first1, last1) and another range starting at *first2*.

The swap operations in the parallel *swap_ranges* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first1* and *last1*.

Template Parameters

- **FwdIter1** – The type of the first range of iterators to swap (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the second range of iterators to swap (deduced). This iterator type must meet the requirements of a forward iterator.

[604] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Parameters

- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.

Returns The *swap_ranges* algorithm returns *FwdIter2*. The *swap_ranges* algorithm returns iterator to the element past the last element exchanged in the range beginning with *first2*.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **swap_ranges**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2)

Exchanges elements between range [first1, last1) and another range starting at *first2*. Executed according to the policy.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first1* and *last1*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the swap operations.
- **FwdIter1** – The type of the first range of iterators to swap (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the second range of iterators to swap (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.

Returns The *swap_ranges* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *parallel_task_policy* and returns *FwdIter2* otherwise. The *swap_ranges* algorithm returns iterator to the element past the last element exchanged in the range beginning with *first2*.

hpx::transform

Defined in header hpx/algorithm.hpp[605].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter1**, typename **FwdIter2**, typename **F**>
FwdIter2 **transform**(*FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *F* &&f)

Applies the given function *f* to the range [first, last) and stores the result in another range, beginning at dest.

Note: Complexity: Exactly *last* - *first* applications of *f*

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *FwdIter2* can be dereferenced and assigned a value of type *Ret*.

Returns The *transform* algorithm returns a *FwdIter2*. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

[605] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **F**>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **transform**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *F* &&f)

Applies the given function *f* to the range [first, last) and stores the result in another range, beginning at dest. Executed according to the policy.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *last* - *first* applications of *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *FwdIter2* can be dereferenced and assigned a value of type *Ret*.

Returns The *transform* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *parallel_task_policy* and returns *FwdIter2* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **F**>
FwdIter3 **transform**(*FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter3* dest, *F* &&f)

Applies the given function *f* to pairs of elements from two ranges: one defined by [first1, last1) and the other beginning at first2, and stores the result in another range, beginning at dest.

Note: Complexity: Exactly *last* - *first* applications of *f*

Template Parameters

- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators for the second range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types FwdIter1 and FwdIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter3* can be dereferenced and assigned a value of type *Ret*.

Returns The *transform* algorithm returns a *FwdIter3*. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **F**>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter3*> **transform**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *FwdIter3* dest, *F* &&f)

Applies the given function *f* to pairs of elements from two ranges: one defined by [first1, last1) and the other beginning at first2, and stores the result in another range, beginning at dest. Executed according to the policy.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *last* - *first* applications of *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the source iterators for the second range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types FwdIter1 and FwdIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter3* can be dereferenced and assigned a value of type *Ret*.

Returns The *transform* algorithm returns a *hpx::future<FwdIter3>* if the execution policy is of type *parallel_task_policy* and returns *FwdIter3* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

hpx::transform_exclusive_scan

Defined in header hpx/algorithm.hpp[606].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **OutIter**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type>
OutIter **transform_exclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest, *T* init, *BinOp* &&binary_op, *UnOp* &&unary_op)

Transforms each element in the range [first, last) with *unary_op*, then computes an exclusive prefix sum operation using *binary_op* over the resulting range, with *init* as the initial value, and writes the results to the range beginning at *dest*. "exclusive" means that the i-th input element is not included in the i-th sum. Formally, assigns through each iterator i in [dest, d_first + (last - first)) the value of the generalized noncommutative sum of init, unary_op(*j)… for every j in [first, first + (i - d_first)) over binary_op, where generalized noncommutative sum GNSUM(op, a1, …, a N) is defined as follows:

- if N=1, a1
- if N > 1, op(GNSUM(op, a1, …, aK), GNSUM(op, aM, …, aN)) for any K where 1 < K+1 = M <= N In other words, the summation operations may be performed in arbitrary order, and the behavior is nondeterministic if *binary_op* is not associative.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *unary_op* nor *binary_op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.

[606] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **binary_op** – Binary *FunctionObject* that will be applied to the result of *unary_op*, the results of other *binary_op*, and *init*.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns The *transform_exclusive_scan* algorithm returns a returns *OutIter*. The *transform_exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*FwdIter1*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **transform_exclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *T* init, *BinOp* &&binary_op, *UnOp* &&unary_op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result) - 1))). Executed according to the policy.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *unary_op* nor *binary_op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- `ExPolicy` – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1` – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2` – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- `BinOp` – The type of *binary_op*.
- `UnOp` – The type of *unary_op*.
- `T` – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `policy` – The execution policy to use for the scheduling of the iterations.
- `first` – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last` – Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest` – Refers to the beginning of the destination range.
- `init` – The initial value for the generalized sum.
- `binary_op` – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init*.
- `unary_op` – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns The *transform_exclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *transform_exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::transform_inclusive_scan

Defined in header hpx/algorithm.hpp[607].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **OutIter**, typename **BinOp**, typename **UnOp**>
OutIter **transform_inclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest, *BinOp* &&binary_op, *UnOp* &&unary_op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *binary_op* nor *unary_op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The difference between *inclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

[607] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns The *transform_inclusive_scan* algorithm returns a returns *OutIter*. The *transform_inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **BinOp**, typename **UnOp**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **transform_inclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *BinOp* &&binary_op, *UnOp* &&unary_op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), …, conv(*(first + (i - result)))). Executed according to the policy.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *binary_op* nor *unary_op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The difference between *inclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unnary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.

Returns The *transform_inclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *transform_inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **OutIter**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type>
OutIter **transform_inclusive_scan**(*InIter* first, *InIter* last, *OutIter* dest, *BinOp* &&binary_op, *UnOp* &&unary_op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *binary_op* nor *unary_op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The difference between *inclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the ith input element in the ith sum. If *binary_op* is not mathematically associative, the behavior of *transform_inclusive_scan* may be non-deterministic.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **BinOp** – The type of *binary_op*.
- **UnOp** – The type of *unary_op*.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unnary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.
- **init** – The initial value for the generalized sum.

Returns The *transform_inclusive_scan* algorithm returns a returns *OutIter*. The *transform_inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*FwdIter1*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **transform_inclusive_scan**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *BinOp* &&binary_op, *UnOp* &&unary_op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, conv(*first), …, conv(*(first + (i - result)))). Executed according to the policy.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *binary_op* nor *unary_op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The difference between *inclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the ith input element in the ith sum. If *binary_op* is not mathematically associative, the behavior of *transform_inclusive_scan* may be non-deterministic.

Note: Complexity: O(last - first) applications of each of *binary_op* and *unary_op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`FwdIter1`** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **`FwdIter2`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **`BinOp`** – The type of *binary_op*.
- **`UnOp`** – The type of *unary_op*.
- **`T`** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **`last`** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **`dest`** – Refers to the beginning of the destination range.

- **binary_op** – Binary *FunctionObject* that will be applied in to the result of *unary_op*, the results of other *binary_op*, and *init* if provided.
- **unnary_op** – Unary *FunctionObject* that will be applied to each element of the input range. The return type must be acceptable as input to *binary_op*.
- **init** – The initial value for the generalized sum.

Returns The *transform_inclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *transform_inclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::transform_reduce

Defined in header hpx/algorithm.hpp[608].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **T**, typename **Reduce**, typename **Convert**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))). Executed according to the policy.

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and

[608] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

– 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *parallel_task_policy* and returns *T* otherwise. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **InIter**, typename **T**, typename **Reduce**, typename **Convert**>
T **transform_reduce**(*InIter* first, *InIter* last, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The difference between *transform_reduce* and *accumulate* is that the behavior of *transform_reduce* may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, ..., aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, ..., bK), GENERALIZED_SUM(op, bM, ..., bN)), where:
  - b1, ..., bN may be any permutation of a1, ..., aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns a *T*. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *T* init)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2. Executed according to the policy.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(*last* - *first*) applications each of *reduce* and *transform*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the first source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **T** – The type of the value to be used as return) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the result will be calculated with.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise.

template<typename **InIter1**, typename **InIter2**, typename **T**>
T **transform_reduce**(*InIter1* first1, *InIter1* last1, *InIter2* first2, *T* init)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

Note: Complexity: O(*last* - *first*) applications each of *reduce* and *transform*.

Template Parameters

- **InIter1** – The type of the first source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as return) values (deduced).

Parameters

- **first1** – Refers to the beginning of the first sequence of elements the result will be calculated with.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.

Returns The *transform_reduce* algorithm returns a *T*.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Reduce**, typename **Convert**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2. Executed according to the policy.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(*last* - *first*) applications each of *reduce* and *transform*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the first source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

- **FwdIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **T** – The type of the value to be used as return) values (deduced).
- **Reduce** – The type of the binary function object used for the multiplication operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the result will be calculated with.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.
- **red_op** – Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.
- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to an object for the second argument type of *red_op*.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise.

template<typename **InIter1**, typename **InIter2**, typename **T**, typename **Reduce**, typename **Convert**>
T **transform_reduce**(ExPolicy &&policy, *InIter1* first1, *InIter1* last1, *InIter2* first2, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

Note: Complexity: O(*last* - *first*) applications each of *reduce* and *transform*.

Template Parameters

- **InIter1** – The type of the first source iterators used (deduced). This iterator type must meet the requirements of an input iterator.

- **InIter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as return) values (deduced).
- **Reduce** – The type of the binary function object used for the multiplication operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **first1** – Refers to the beginning of the first sequence of elements the result will be calculated with.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.
- **red_op** – Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.
- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to an object for the second argument type of *red_op*.

Returns The *transform_reduce* algorithm returns a *T*.

hpx/parallel/algorithms/transform_reduce_binary.hpp

Defined in header hpx/parallel/algorithms/transform_reduce_binary.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::uninitialized_copy, hpx::uninitialized_copy_n

Defined in header hpx/algorithm.hpp[609].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **FwdIter**>
FwdIter **uninitialized_copy**(*InIter* first, *InIter* last, *FwdIter* dest)

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_copy* algorithm returns *FwdIter*. The *uninitialized_copy* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **uninitialized_copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects. Executed according to the policy.

[609] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_copy* algorithm returns a *hpx::future<FwdIter2>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_copy* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **Size**, typename **FwdIter**>
FwdIter **uninitialized_copy_n**(*InIter* first, *Size* count, *FwdIter* dest)

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_copy_n* algorithm returns a returns *FwdIter*. The *uninitialized_copy_n* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **uninitialized_copy_n**(*ExPolicy* &&policy, *FwdIter1* first, *Size* count, *FwdIter2* dest)

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_copy_n* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_copy_n* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::uninitialized_default_construct, hpx::uninitialized_default_construct_n

Defined in header hpx/algorithm.hpp[610].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**>
void **uninitialized_default_construct**(*FwdIter* first, *FwdIter* last)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *uninitialized_default_construct* algorithm returns nothing

template<typename **ExPolicy**, typename **FwdIter**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **uninitialized_default_construct**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects. Executed according to the policy.

[610] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *uninitialized_default_construct* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

template<typename **FwdIter**, typename **Size**>
FwdIter **uninitialized_default_construct_n**(*FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range [first, first + count) by default-initialization. If an exception is thrown during the initialization, the function has no effects.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_default_construct_n* algorithm returns a returns *FwdIter*. The *uninitialized_default_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_default_construct_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range [first, first + count) by default-initialization. If an exception is thrown during the initialization, the function has no effects. Executed according to the policy.

The assignments in the parallel *uninitialized_default_construct_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_default_construct_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_default_construct_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_default_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

hpx::uninitialized_fill, hpx::uninitialized_fill_n

Defined in header hpx/algorithm.hpp[611].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[611] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **FwdIter**, typename **T**>
void **uninitialized_fill**(*FwdIter* first, *FwdIter* last, *T* const &value)

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

Note: Complexity: Linear in the distance between *first* and *last*

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *uninitialized_fill* algorithm returns nothing

template<typename **ExPolicy**, typename **FwdIter**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **uninitialized_fill**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *T* const &value)

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects. Executed according to the policy.

The initializations in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The initializations in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *uninitialized_fill* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

template<typename **FwdIter**, typename **Size**, typename **T**>
FwdIter **uninitialized_fill_n**(*FwdIter* first, *Size* count, *T* const &value)

Copies the given *value* value to the first count elements in an uninitialized memory area beginning at first. If an exception is thrown during the initialization, the function has no effects.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *uninitialized_fill_n* algorithm returns a returns *FwdIter*. The *uninitialized_fill_n* algorithm returns the output iterator to the element in the range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_fill_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *T* const &value)

Copies the given *value* value to the first count elements in an uninitialized memory area beginning at first. If an exception is thrown during the initialization, the function has no effects. Executed according to the policy.

The initializations in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The initializations in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *uninitialized_fill_n* algorithm returns a *hpx::future<FwdIter>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns FwdIter otherwise. The *uninitialized_fill_n* algorithm returns the output iterator to the element in the range, one past the last element copied.

hpx::uninitialized_move, hpx::uninitialized_move_n

Defined in header hpx/algorithm.hpp[612].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **InIter**, typename **FwdIter**>
FwdIter **uninitialized_move**(*InIter* first, *InIter* last, *FwdIter* dest)

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.

[612] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_move* algorithm returns *FwdIter*. The *uninitialized_move* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **uninitialized_move**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest)

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state. Executed according to the policy.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_move* algorithm returns a *hpx::future<FwdIter2>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_move* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

template<typename **InIter**, typename **Size**, typename **FwdIter**>
std::pair<*InIter*, *FwdIter*> **uninitialized_move_n**(*InIter* first, *Size* count, *FwdIter* dest)

Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state.

Note: Complexity: Performs exactly *count* movements, if count > 0, no move operations otherwise.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_move_n* algorithm returns a returns *std::pair<InIter,FwdIter>*. The *uninitialized_move_n* algorithm returns A pair whose first element is an iterator to the element past the last element moved in the source range, and whose second element is an iterator to the element past the last element moved in the destination range.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *std*::pair<*FwdIter1*, *FwdIter2*>>::type **uninitialized_move_n**(*ExPolicy* &&policy, *FwdIter1* first, *Size* count, *FwdIter2* dest)

Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state. Executed according to the policy.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* movements, if count > 0, no move operations otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_move_n* algorithm returns a *hpx::future<std::pair<FwdIter1,FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1,FwdIter2>* otherwise. The *uninitialized_move_n* algorithm returns A pair whose first element is an iterator to the element past the last element moved in the source range, and whose second element is an iterator to the element past the last element moved in the destination range.

hpx::unititiallized_relocate, hpx::uninitialized_relocate_n

Defined in header hpx/algorithm.hpp[613].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[613] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **InIter1**, typename **InIter2**, typename **FwdIter**>
FwdIter **uninitialized_relocate**(*InIter1* first, *InIter2* last, *FwdIter* dest)

Relocates the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the move-construction of an element, all elements left in the input range, as well as all objects already constructed in the destination range are destroyed. After this algorithm completes, the source range should be freed or reused without destroying the objects.

The assignments in the parallel *uninitialized_relocate* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: time: O(n), space: O(1) 1) For “trivially relocatable” underlying types (T) and a contiguous iterator range [first, last): std::distance(first, last)*sizeof(T) bytes are copied. 2) For “trivially relocatable” underlying types (T) and a non-contiguous iterator range [first, last): std::distance(first, last) memory copies of sizeof(T) bytes each are performed. 3) For “non-trivially relocatable” underlying types (T): std::distance(first, last) move assignments and destructions are performed.

Note: Declare a type as “trivially relocatable” using the `HPX_DECLARE_TRIVIALLY_RELOCATABLE` macros found in <hpx/type_support/is_trivially_relocatable.hpp>.

Template Parameters

- **InIter1** – The type of the source iterator first (deduced). This iterator type must meet the requirements of an input iterator.
- **InIter2** – The type of the source iterator last (deduced). This iterator type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_relocate* algorithm returns *FwdIter*. The *uninitialized_relocate* algorithm returns the output iterator to the element in the destination range, one past the last element relocated.

template<typename **ExPolicy**, typename **InIter1**, typename **InIter2**, typename **FwdIter**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_relocate**(*ExPolicy* &&policy, *InIter1* first, *InIter2* last, *FwdIter* dest)

Relocates the elements in the range defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the move-construction of an element, all elements left in the input range, as well as all objects already constructed in the destination range are destroyed. After this algorithm completes, the source range should be freed or reused without destroying the objects.

The assignments in the parallel *uninitialized_relocate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: time: O(n), space: O(1) 1) For "trivially relocatable" underlying types (T) and a contiguous iterator range [first, last): std::distance(first, last)*sizeof(T) bytes are copied. 2) For "trivially relocatable" underlying types (T) and a non-contiguous iterator range [first, last): std::distance(first, last) memory copies of sizeof(T) bytes each are performed. 3) For "non-trivially relocatable" underlying types (T): std::distance(first, last) move assignments and destructions are performed.

Note: Declare a type as "trivially relocatable" using the `HPX_DECLARE_TRIVIALLY_RELOCATABLE` macros found in <hpx/type_support/is_trivially_relocatable.hpp>.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`InIter1`** – The type of the source iterator first (deduced). This iterator type must meet the requirements of an input iterator.
- **`InIter2`** – The type of the source iterator last (deduced). This iterator type must meet the requirements of an input iterator.
- **`FwdIter`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **`last`** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **`dest`** – Refers to the beginning of the destination range. The assignments in the parallel *uninitialized_relocate_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Returns The *uninitialized_relocate* algorithm returns a *hpx::future<FwdIter>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_relocate* algorithm returns the output iterator to the element in the destination range, one past the last element relocated.

template<typename **BiIter1**, typename **BiIter2**>
BiIter2 **`uninitialized_relocate_backward`**(*BiIter1* first, *BiIter1* last, *BiIter2* dest_last)

Relocates the elements in the range, defined by [first, last), to an uninitialized memory area ending at *dest_last*. The objects are processed in reverse order. If an exception is thrown during the the move-construction of an element, all elements left in the input range, as well as all objects already constructed in the destination range are destroyed. After this algorithm completes, the source range should be freed or reused without destroying the objects.

The assignments in the parallel *uninitialized_relocate* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: time: O(n), space: O(1) 1) For "trivially relocatable" underlying types (T) and a contiguous iterator range [first, last): std::distance(first, last)*sizeof(T) bytes are copied. 2) For "trivially relocatable" underlying types (T) and a non-contiguous iterator range [first, last): std::distance(first, last) memory copies of sizeof(T) bytes each are performed. 3) For "non-trivially relocatable" underlying types (T): std::distance(first, last) move assignments and destructions are performed.

Note: Declare a type as "trivially relocatable" using the `HPX_DECLARE_TRIVIALLY_RELOCATABLE` macros found in <hpx/type_support/is_trivially_relocatable.hpp>.

Template Parameters

- **BiIter1** – The type of the source range (deduced). This iterator type must meet the requirements of a Bidirectional iterator.
- **BiIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a Bidirectional iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest_last** – Refers to the beginning of the destination range.

Returns The *uninitialized_relocate_backward* algorithm returns *BiIter2*. The *uninitialized_relocate_backward* algorithm returns the bidirectional iterator to the first element in the destination range.

template<typename **ExPolicy**, typename **BiIter1**, typename **BiIter2**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *BiIter2*> **uninitialized_relocate_backward**(*ExPolicy* &&policy, *BiIter1* first, *BiIter1* last, *BiIter2* dest_last)

Relocates the elements in the range, defined by [first, last), to an uninitialized memory area ending at *dest_last*. The order of the relocation of the objects depends on the execution policy. If an exception is thrown during the the move-construction of an element, all elements left in the input range, as well as all objects already constructed in the destination range are destroyed. After this algorithm completes, the source range should be freed or reused without destroying the objects.

The assignments in the parallel *uninitialized_relocate_backward* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Using the *uninitialized_relocate_backward* algorithm with the with a non-sequenced execution policy, will not guarantee the order of the relocation of the objects.

Note: Complexity: time: O(n), space: O(1) 1) For "trivially relocatable" underlying types (T) and a contiguous iterator range [first, last): std::distance(first, last)*sizeof(T) bytes are copied. 2) For "trivially relocatable" underlying types (T) and a non-contiguous iterator range [first, last): std::distance(first, last) memory copies of sizeof(T) bytes each are performed. 3) For "non-trivially relocatable" underlying types (T): std::distance(first, last) move assignments and destructions are performed.

Note: Declare a type as "trivially relocatable" using the `HPX_DECLARE_TRIVIALLY_RELOCATABLE` macros found in <hpx/type_support/is_trivially_relocatable.hpp>.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`BiIter1`** – The type of the source range (deduced). This iterator type must meet the requirements of a Bidirectional iterator.
- **`BiIter2`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a Bidirectional iterator.

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **`last`** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **`dest_last`** – Refers to the end of the destination range.

Returns The *uninitialized_relocate_backward* algorithm returns a *hpx::future<FwdIter>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *BiIter2* otherwise. The *uninitialized_relocate_backward* algorithm returns the bidirectional iterator to the first element in the destination range.

template<typename **InIter**, typename **Size**, typename **FwdIter**>
FwdIter **uninitialized_relocate_n**(*InIter* first, *Size* count, *FwdIter* dest)

Relocates the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the move-construction of an element, all elements left in the input range, as well as all objects already constructed in the destination range are destroyed. After this algorithm completes, the source range should be freed or reused without destroying the objects.

The assignments in the parallel *uninitialized_relocate_n* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: time: O(n), space: O(1) 1) For "trivially relocatable" underlying types (T) and a contiguous iterator range [first, first+count): `count*sizeof(T)` bytes are copied. 2) For "trivially relocatable" underlying types (T) and a non-contiguous iterator range [first, first+count): `count` memory copies of sizeof(T) bytes each are performed. 3) For "non-trivially relocatable" underlying types (T): `count` move assignments and destructions are performed.

Note: Declare a type as "trivially relocatable" using the `HPX_DECLARE_TRIVIALLY_RELOCATABLE` macros found in <hpx/type_support/is_trivially_relocatable.hpp>.

Template Parameters

- **InIter** – The type of the source iterator first (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to relocate.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *uninitialized_relocate_n* algorithm returns *FwdIter*. The *uninitialized_relocate_n* algorithm returns the output iterator to the element in the destination range, one past the last element relocated.

template<typename **ExPolicy**, typename **InIter**, typename **Size**, typename **FwdIter**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_relocate_n**(*ExPolicy* &&policy, *InIter* first, *Size* count, *FwdIter* dest)

Relocates the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the move-construction of an element, all elements left in the input range, as well as all objects already constructed in the destination range are destroyed. After this algorithm completes, the source range should be freed or reused without destroying the objects.

The assignments in the parallel *uninitialized_relocate_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_relocate_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: time: O(n), space: O(1) 1) For "trivially relocatable" underlying types (T) and a contiguous iterator range [first, first+count): `count*sizeof(T)` bytes are copied. 2) For "trivially relocatable" underlying types (T) and a non-contiguous iterator range [first, first+count): `count` memory copies of sizeof(T) bytes each are performed. 3) For "non-trivially relocatable" underlying types (T): `count` move assignments and destructions are performed.

Note: Declare a type as "trivially relocatable" using the `HPX_DECLARE_TRIVIALLY_RELOCATABLE` macros found in <hpx/type_support/is_trivially_relocatable.hpp>.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`InIter`** – The type of the source iterator first (deduced). This iterator type must meet the requirements of an input iterator.
- **`Size`** – The type of the argument specifying the number of elements to relocate.
- **`FwdIter`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **`count`** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **`dest`** – Refers to the beginning of the destination range.

Returns The *uninitialized_relocate_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_relocate_n* algorithm returns the output iterator to the element in the destination range, one past the last element relocated.

hpx::uninitialized_value_construct, hpx::uninitialized_value_construct_n

Defined in header hpx/algorithm.hpp[614].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[614] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **FwdIter**>
void **uninitialized_value_construct**(*FwdIter* first, *FwdIter* last)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *uninitialized_value_construct* algorithm returns nothing

template<typename **ExPolicy**, typename **FwdIter**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **uninitialized_value_construct**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects. Executed according to the policy.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *uninitialized_value_construct* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

template<typename **FwdIter**, typename **Size**>
FwdIter **uninitialized_value_construct_n**(*FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_value_construct_n* algorithm returns a returns *FwdIter*. The *uninitialized_value_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_value_construct_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_value_construct_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_value_construct_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_value_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

hpx::unique, hpx::unique_copy

Defined in header hpx/algorithm.hpp[615].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **FwdIter**, typename **Pred** = *hpx*::*parallel*::detail::equal_to, typename **Proj** = *hpx*::identity>
FwdIter **unique**(*FwdIter* first, *FwdIter* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

[615] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *pred* is invoked.

Returns The *unique* algorithm returns *FwdIter*. The *unique* algorithm returns the iterator to the new end of the range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = *hpx*::*parallel*::detail::equal_to, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **unique**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range. Executed according to the policy.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.

- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *pred* is invoked.

Returns The *unique* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *unique* algorithm returns the iterator to the new end of the range.

template<typename **InIter**, typename **OutIter**, typename **Pred** = *hpx*::*parallel*::detail::equal_to, typename **Proj** = *hpx*::identity>
OutIter **unique_copy**(*InIter* first, *InIter* last, *OutIter* dest, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *pred* is invoked.

Returns The *unique_copy* algorithm returns a returns *OutIter*. The *unique_copy* algorithm returns the destination iterator to the end of the *dest* range.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = *hpx*::*parallel*::detail::equal_to, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **unique_copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied. Executed according to the policy.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *pred* is invoked.

Returns The *unique_copy* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

hpx::ranges::adjacent_difference

Defined in header hpx/algorithm.hpp[616].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter1**, typename **FwdIter2**, typename **Sent**>
FwdIter2 **adjacent_difference**(*FwdIter1* first, *Sent* last, *FwdIter2* dest)

Searches the range [first, last) for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters

- **FwdIter1** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.

[616] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **Rng**, typename **FwdIter2**>
FwdIter2 **adjacent_difference**(*Rng* &&rng, *FwdIter2* dest)

Searches the *rng* for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters

- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **adjacent_difference**(*ExPolicy* &&policy, *FwdIter1* first, *Sent* last, *FwdIter2* dest)

Searches the range [first, last) for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **adjacent_difference**(*ExPolicy* &&policy, *Rng* &&rng, *FwdIter2* dest)

Searches the *rng* for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

template<typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **Op**>
FwdIter2 **adjacent_difference**(*FwdIter1* first, *Sent* last, *FwdIter2* dest, *Op* &&op)

Searches the range [first, last) for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters
- **FwdIter1** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **Op** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *adjacent_difference* requires *Op* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &. The types *Type1* and *Type2* must be such that an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **Rng**, typename **FwdIter2**, typename **Op**>
FwdIter2 **adjacent_difference**(*Rng* &&rng, *FwdIter2* dest, *Op* &&op)

Searches the *rng* for two consecutive identical elements.

Template Parameters

- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Op** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *adjacent_difference* requires *Op* to meet the requirements of *CopyConstructible*.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &. The types *Type1* and *Type2* must be such that an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.?

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **Op**>

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **adjacent_difference**(*ExPolicy* &&policy, *FwdIter1* first, *Sent* last, *FwdIter2* dest, *Op* &&op)

Searches the range [first, last) for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **Op** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *adjacent_difference* requires *Op* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &. The types *Type1* and *Type2* must be such that an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.?

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter2**, typename **Op**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **adjacent_difference**(*ExPolicy* &&policy, *Rng* &&rng, *FwdIter2* dest, *Op* &&op)

Searches the *rng* for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

---

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter2** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Op** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *adjacent_difference* requires *Op* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Binary operation function object that will be applied. The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &. The types *Type1* and *Type2* must be such that an object of type *iterator_traits<InputIt>::value_type* can be implicitly converted to both of them. The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *adjacent_difference* algorithm returns an iterator to the first of the identical elements.

hpx::ranges::adjacent_find

Defined in header hpx/algorithm.hpp[617].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[617] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **FwdIter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **Pred** = detail::equal_to>
FwdIter **adjacent_find**(*FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Searches the range [first, last) for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

Template Parameters

- **FwdIter** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*
- **Pred** – The type of an optional function/function object to use.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **Pred** = detail::equal_to>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **adjacent_find**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Searches the range [first, last) for two consecutive identical elements. This version uses the given binary predicate pred

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *adjacent_find* is available if the user decides to provide their algorithm their own binary predicate *pred*.

Note: Complexity: Exactly the smaller of (result - first) + 1 and (last - first) - 1 application of the predicate where *result* is the value returned

---

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *adjacent_find* algorithm returns a *hpx::future<InIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *InIter* otherwise. The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **Rng**, typename **Proj** = *hpx*::identity, typename **Pred** = detail::equal_to>
hpx::*traits*::range_traits<*Rng*>::iterator_type **adjacent_find**(*Rng* &&rng, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Searches the range rng for two consecutive identical elements.

Note: Complexity: Exactly the smaller of (result - std::begin(rng)) + 1 and (std::begin(rng) - std::end(rng)) - 1 applications of the predicate where *result* is the value returned

---

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*
- **Pred** – The type of an optional function/function object to use.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

template<typename **ExPolicy**, typename **Rng**, typename **Proj** = *hpx*::identity, typename **Pred** = detail::equal_to>
parallel::*util*::detail::algorithm_result<*ExPolicy*, typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::type **adjacent_**

Searches the range rng for two consecutive identical elements.

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *adjacent_find* invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *adjacent_find* is available if the user decides to provide their algorithm their own binary predicate *pred*.

Note: Complexity: Exactly the smaller of (result - std::begin(rng)) + 1 and (std::begin(rng) - std::end(rng)) - 1 applications of the predicate where *result* is the value returned

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *adjacent_find* algorithm returns a *hpx::future<InIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *InIter* otherwise. The *adjacent_find* algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is returned.

hpx::ranges::all_of, hpx::ranges::any_of, hpx::ranges::none_of

Defined in header hpx/algorithm.hpp[618].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **none_of**(*ExPolicy* &&policy, *Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for no elements in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

[618] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *none_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **none_of**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for no elements in the range [first, last).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *last* - *first* applications of the predicate *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential

form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.

- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *none_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

template<typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
bool **none_of**(*Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for no elements in the range *rng*.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

template<typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
bool **none_of**(*Iter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for no elements in the range [first, last).

Note: Complexity: At most *last* - *first* applications of the predicate *f*

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *none_of* algorithm returns true if the unary predicate *f* returns true for no elements in the range, false otherwise. It returns true if the range is empty.

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **any_of**(*ExPolicy* &&policy, *Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *any_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **any_of**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *any_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

template<typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
bool **any_of**(*Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

template<typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
bool **any_of**(*Iter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *any_of* algorithm returns true if the unary predicate *f* returns true for at least one element in the range, false otherwise. It returns false if the range is empty.

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **all_of**(*ExPolicy* &&policy, *Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for all elements in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *Copy-*

Constructible.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *all_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **all_of**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for all elements in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.

- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *all_of* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

template<typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
bool **all_of**(*Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for all elements in the range *rng*.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

template<typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
bool **all_of**(*Iter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Checks if unary predicate *f* returns true for all elements in the range *rng*.

Note: Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate *f*

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *none_of* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *all_of* algorithm returns true if the unary predicate *f* returns true for all elements in the range, false otherwise. It returns true if the range is empty.

hpx::ranges::copy, hpx::ranges::copy_n, hpx::ranges::copy_if

Defined in header hpx/algorithm.hpp[619].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::copy_result<*FwdIter1*, *FwdIter*>>::type **copy**(*ExPolicy* &&policy, *FwdIter1* iter, *Sent1* sent, *FwdIter* dest)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*.

[619] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **iter** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns a *hpx::future<ranges::copy_result<FwdIter1, FwdIter> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_result<FwdIter1, FwdIter>* otherwise. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::copy_result<typename *hpx*::*traits*::range_traits<*Rng*>::iterator_typ

Copies the elements in the range *rng* to another range beginning at *dest*.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns a *hpx::future<ranges::copy_result<iterator_t<Rng>*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_result<iterator_t<Rng>*, FwdIter2> otherwise. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **Sent1**, typename **FwdIter**>
ranges::copy_result<*FwdIter1*, *FwdIter*> **copy**(*FwdIter1* iter, *Sent1* sent, *FwdIter* dest)

Copies the elements in the range, defined by [first, last), to another range beginning at *dest*.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **iter** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **FwdIter**>
ranges::copy_result<typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type, *FwdIter*> **copy**(*Rng* &&rng, *FwdIter* dest)

Copies the elements in the range *rng* to another range beginning at *dest*.

Note: Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::copy_n_result<*FwdIter1*, *FwdIter2*>>::type **copy_n**(*ExPolicy* &&policy, *FwdIter1* first, *Size* count, *FwdIter2* dest)

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest.

The assignments in the parallel *copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy_n* algorithm returns a *hpx::future<ranges::copy_n_result<FwdIter1, FwdIter2> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_n_result<FwdIter1, FwdIter2>* otherwise. The *copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **Size**, typename **FwdIter2**>

ranges::copy_n_result<*FwdIter1*, *FwdIter2*> **copy_n**(*FwdIter1* first, *Size* count, *FwdIter2* dest)

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::copy_if_result<*FwdIter1*, *FwdIter*>>::type **copy_if**(*ExPolic* &&pol- icy, *FwdIte* iter, *Sent1* sent, *FwdIte* dest, *Pred* &&pre *Proj* &&pro = *Proj*())

Copies the elements in the range, defined by [first, last) to another range beginning at *dest*. The order of the elements that are not removed is preserved.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for FwdIter1.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **iter** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *copy_if* algorithm returns a *hpx::future<ranges::copy_if_result<iterator_t<Rng>*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_if_result<iterator_t<Rng>*, FwdIter2> otherwise. The *copy_if* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::copy_if_result<typename *hpx*::*traits*::range_traits<*Rng*>::itera

Copies the elements in the range, defined by *rng* to another range beginning at *dest*. The order of the elements that are not removed is preserved.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *copy_if* algorithm returns a *hpx::future<ranges::copy_if_result<iterator_t<Rng>*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_if_result<iterator_t<Rng>*, FwdIter2> otherwise. The *copy_if* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **Sent1**, typename **FwdIter**, typename **Pred**, typename **Proj** = *hpx*::identity>
ranges::copy_if_result<*FwdIter1*, *FwdIter*> **copy_if**(*FwdIter1* iter, *Sent1* sent, *FwdIter* dest, *Pred* &&pred, *Proj* &&proj = *Proj*())

Copies the elements in the range, defined by [first, last) to another range beginning at *dest*. The order of the elements that are not removed is preserved.

Template Parameters

- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for FwdIter1.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **iter** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *copy_if* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **FwdIter**, typename **Pred**, typename **Proj** = *hpx*::identity>
ranges::copy_if_result<typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type, *FwdIter*> **copy_if**(*Rng* &&rng, *FwdIter* dest, *Pred* &&pred, *Proj* &&proj = *Proj*())

Copies the elements in the range, defined by *rng* to another range beginning at *dest*. The order of the elements that are not removed is preserved.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1* .

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *copy_if* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::count, hpx::ranges::count_if

Defined in header hpx/algorithm.hpp[620].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Rng**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<typename *hpx*::*traits*::range_traits<*Rng*>

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* comparisons.

Note: The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to search for (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

[620] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns a *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<*Iter*>::difference_type>::type **count**(*Ex*
&&
icy
Ite
fir
Se
las
T
co
&v
Pr
&&
=
Pr

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* comparisons.

Note: The comparisons in the parallel *count* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **T** – The type of the value to search for (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns a *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **Rng**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>
std::iterator_traits<typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::difference_type **count**(*Rng* &&rng, *T* const &value, *Proj* &&proj = *Proj*())

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

Note: Complexity: Performs exactly *last* - *first* comparisons.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to search for (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type>
std::iterator_traits<*Iter*>::difference_type **count**(*Iter* first, *Sent* last, *T* const &value, *Proj* &&proj = *Proj*())

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts the elements that are equal to the given *value*.

Note: Complexity: Performs exactly *last* - *first* comparisons.

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **T** – The type of the value to search for (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – The value to search for.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<typename *hpx*::*traits*::range_traits<*Rng*>

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Note: The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count_if* algorithm returns *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<*Iter*>::difference_type>::type **count_if**

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Note: The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: The assignments in the parallel *count_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the comparisons.
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *Copy-*

Constructible.

- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count_if* algorithm returns *hpx::future<difference_type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *std::iterator_traits<FwdIter>::difference_type*. The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
std::iterator_traits<typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::difference_type **count_if**(*Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

template<typename **Iter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
std::iterator_traits<*Iter*>::difference_type **count_if**(*Iter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Returns the number of elements in the range [first, last) satisfying a specific criteria. This version counts elements for which predicate *f* returns true.

Note: Complexity: Performs exactly *last* - *first* applications of the predicate.

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *count_if* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *count* algorithm returns the number of elements satisfying the given criteria.

hpx::ranges::destroy, hpx::ranges::destroy_n

Defined in header hpx/algorithm.hpp[621].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[621] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Rng**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **destroy**(*ExPolicy* &&policy, *Rng* &&rng)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

The operations in the parallel *destroy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *destroy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* operations.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.

Returns The *destroy* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **destroy**(*ExPolicy* &&policy, *Iter* first, *Sent* last)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

The operations in the parallel *destroy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *destroy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* operations.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used for the range (deduced).

- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *destroy* algorithm returns a *hpx::future<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **Rng**>
hpx::*traits*::range_iterator<*Rng*>::type **destroy**(*Rng* &&rng)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

Note: Complexity: Performs exactly *last* - *first* operations.

Template Parameters **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters **rng** – Refers to the sequence of elements the algorithm will be applied to.

Returns The *destroy* algorithm returns *void*.

template<typename **Iter**, typename **Sent**>
Iter **destroy**(*Iter* first, *Sent* last)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, last).

Note: Complexity: Performs exactly *last* - *first* operations.

Template Parameters

- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *destroy* algorithm returns *void*.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **destroy_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, first + count).

The operations in the parallel *destroy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *destroy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* operations, if count > 0, no assignments otherwise.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply this algorithm to.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *destroy_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **FwdIter**, typename **Size**>
FwdIter **destroy_n**(*FwdIter* first, *Size* count)

Destroys objects of type typename iterator_traits<ForwardIt>::value_type in the range [first, first + count).

Note: Complexity: Performs exactly *count* operations, if count > 0, no assignments otherwise.

Template Parameters
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply this algorithm to.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *destroy_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

hpx::ranges::ends_with

Defined in header hpx/algorithm.hpp[622].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[622] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

namespace **ranges**

Functions

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **ends_with**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range defined by [first1, last1) matches the suffix of the first range defined by [first2, last2)

The assignments in the parallel *ends_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters
- **Iter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters
- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *ends_with* algorithm returns *bool*. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, bool>::type **ends_with**(*ExPolicy* &&policy, *FwdIter1* first1, *Sent1* last1, *FwdIter2* first2, *Sent2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range defined by [first1, last1) matches the suffix of the first range defined by [first2, last2)

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **FwdIter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *ends_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **ends_with**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range *rng2* matches the suffix of the first range *rng1*.

The assignments in the parallel *ends_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *ends_with* algorithm returns *bool*. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, bool>::type **ends_with**(*ExPolicy* &&policy, *Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range *rng2* matches the suffix of the first range *rng1*.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *ends_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *ends_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *ends_with* algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

hpx::ranges::equal

Defined in header hpx/algorithm.hpp[623].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[623] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **equal**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *f*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the source iterators used for the end of the first range (deduced).
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the source iterators used for the end of the second range (deduced).
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.

- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *equal* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **equal**(*ExPolicy* &&policy, *Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if the range [first1, last1) is equal to the range starting at first2, and false otherwise.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *last1* - *first1* applications of the predicate *f*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

- **Proj1** – The type of an optional projection function applied to the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *equal* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **equal**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *f*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **Iter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the source iterators used for the end of the first range (deduced).
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the source iterators used for the end of the second range (deduced).
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **equal**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if the range [first1, last1) is equal to the range starting at first2, and false otherwise.

Note: Complexity: At most *last1* - *first1* applications of the predicate *f*.

Note: The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

Template Parameters

- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.

- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *equal* algorithm returns true if the elements in the two ranges are equal, otherwise it returns false.

hpx::ranges::exclusive_scan

Defined in header hpx/algorithm.hpp[624].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type, typename **Op** = *std*::plus<*T*>>
exclusive_scan_result<*InIter*, *OutIter*> **exclusive_scan**(*InIter* first, *Sent* last, *OutIter* dest, *T* init, *Op* &&op = *Op*())

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, …, *(first + (i - result) - 1)).

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

[624] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Template Parameters

- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter1.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Op** – The type of the binary function object used for the reduction operation.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **T** = typename *std*::iterator_traits<*FwdIter1*>::value_type, typename **Op** = *std*::plus<*T*>>
parallel::*util*::detail::algorithm_result<*ExPolicy*, exclusive_scan_result<*FwdIter1*, *FwdIter2*>>::type **exclusive_scan**(*Ex* && icy *Fw* firs *Ser* las *Fw* des *T* init *Op* && = *Op*

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, …, *(first + (i - result) - 1)).

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in

unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter1.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Op** – The type of the binary function object used for the reduction operation.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *exclusive_scan* algorithm returns a *hpx::future<util::in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result<FwdIter1, FwdIter2>* otherwise. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **O**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type, typename **Op** = *std*::plus<*T*>>
exclusive_scan_result<*traits*::range_iterator_t<*Rng*>, *O*> **exclusive_scan**(*Rng* &&rng, *O* dest, *T* init, *Op* &&op = *Op*())

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, …, *(first + (i - result) - 1))

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *std::plus<T>*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:
- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Op** – The type of the binary function object used for the reduction operation.

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **O**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type, typename **Op** = *std*::plus<*T*>>

parallel::*util*::detail::algorithm_result<*ExPolicy*, exclusive_scan_result<*traits*::range_iterator_t<*Rng*>, *O*>> **exclusive_sc**

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, …, *(first + (i - result) - 1))

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *std::plus<T>*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Op** – The type of the binary function object used for the reduction operation.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate

should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *exclusive_scan* algorithm returns a *hpx::future<util::in_out_result* <traits::range_iterator_t<Rng>, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result* <traits::range_iterator_t<Rng>, O> otherwise. The *exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::fill, hpx::ranges::fill_n

Defined in header hpx/algorithm.hpp[625].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Rng**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type> *hpx*::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::type **fill**(

Assigns the given value to the elements in the range [first, last).

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

[625] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **T** = typename *std*::iterator_traits<*Iter*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *Iter*>::type **fill**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *T* const &value)

Assigns the given value to the elements in the range [first, last).

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

template<typename **Rng**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
hpx::*traits*::range_iterator_t<*Rng*> **fill**(*Rng* &&rng, *T* const &value)

Assigns the given value to the elements in the range [first, last).

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns *void*.

template<typename **Iter**, typename **Sent**, typename **T** = typename *std*::iterator_traits<*Iter*>::value_type>
Iter **fill**(*Iter* first, *Sent* last, *T* const &value)

Assigns the given value to the elements in the range [first, last).

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters
- **Iter** – The type of the source iterators used for the range (deduced).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **T** – The type of the value to be assigned (deduced).

Parameters
- **first** – Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill* algorithm returns *void*.

template<typename **ExPolicy**, typename **Rng**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **fill_n**(*ExPolicy* &&policy, *Rng* &&rng, *T* const &value)

Assigns the given value value to the first count elements in the range beginning at first if count > 0. Does nothing otherwise.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **fill_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *T* const &value)

Assigns the given value value to the first count elements in the range beginning at first if count > 0. Does nothing otherwise.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, for count > 0.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

template<typename **Rng**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>

hpx::*traits*::range_traits<*Rng*>::iterator_type **fill_n**(*Rng* &&rng, *T* const &value)

Assigns the given value value to the first count elements in the range beginning at first if count > 0. Does nothing otherwise.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill_n* algorithm returns an output iterator that compares equal to last.

template<typename **FwdIter**, typename **Size**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
FwdIter **fill_n**(Iterator first, *Size* count, *T* const &value)

Assigns the given value value to the first count elements in the range beginning at first if count > 0. Does nothing otherwise.

Note: Complexity: Performs exactly *count* assignments, for count > 0.

Template Parameters

- **Iterator** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *fill_n* algorithm returns an output iterator that compares equal to last.

hpx::ranges::find, hpx::ranges::find_if, hpx::ranges::find_if_not, hpx::ranges::find_end, hpx::ranges::find_first_of

Defined in header hpx/algorithm.hpp[626].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[626] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **find**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *T* const &val, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **T** – The type of the value to find (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **val** – the value to compare the elements to
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **find**(*ExPolicy* &&policy, *Rng* &&rng, *T* const &val, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to find (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **val** – the value to compare the elements to
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

template<typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type>
Iter **find**(*Iter* first, *Sent* last, *T* const &val, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) that is equal to value

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **T** – The type of the value to find (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **val** – the value to compare the elements to
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

template<typename **Rng**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>
hpx::*traits*::range_iterator_t<*Rng*> **find**(*Rng* &&rng, *T* const &val, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the operator==().

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to find (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **val** – the value to compare the elements to
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find* algorithm returns the first element in the range [first,last) that is equal to *val*. If no such element in the range of [first,last) is equal to *val*, then the algorithm returns *last*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **find_if**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) for which predicate *pred* returns true

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **pred** – The unary predicate which returns true for the required element. The signature of the predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if* algorithm returns the first element in the range [first,last) that satisfies the predicate *f*. If no such element exists that satisfies the predicate f, the algorithm returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **find_if**(*ExPolicy* &&policy, *Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Returns the first element in the range *rng* for which predicate *pred* returns true

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – The unary predicate which returns true for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if* algorithm returns the first element in the range [first,last) that satisfies the predicate *f*. If no such element exists that satisfies the predicate f, the algorithm returns *last*.

template<typename **Iter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
Iter **find_if**(*Iter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) for which predicate *pred* returns true

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.

- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **pred** – The unary predicate which returns true for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if* algorithm returns the first element in the range [first,last) that satisfies the predicate *f*. If no such element exists that satisfies the predicate f, the algorithm returns *last*.

template<typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **find_if**(*Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Returns the first element in the range *rng* for which predicate *pred* returns true

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – The unary predicate which returns true for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if* algorithm returns the first element in the range [first,last) that satisfies the predicate *f*. If no such element exists that satisfies the predicate f, the algorithm returns *last*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *Iter*>::type **find_if_not**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) for which predicate *f* returns false

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **pred** – The unary predicate which returns false for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if_not* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if_not* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate f, the algorithm returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>

hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng>> **find_if_not**(ExPolicy &&policy, Rng &&rng, Pred &&pred, Proj &&proj = Proj())

Returns the first element in the range *rng* for which predicate *f* returns false

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_if_not* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – The unary predicate which returns false for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if_not* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *find_if_not* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate f, the algorithm returns *last*.

template<typename **Iter**, typename **Sent**, typename **Pred**, typename **Proj** = hpx::identity>

Iter **find_if_not**(*Iter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) for which predicate *f* returns false

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **Iter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **pred** – The unary predicate which returns false for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if_not* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate f, the algorithm returns *last*.

template<typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **find_if_not**(*Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Returns the first element in the range *rng* for which predicate *f* returns false

Note: Complexity: At most last - first applications of the predicate.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – The unary predicate which returns false for the required element. The signature of the predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *find_if_not* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate f, the algorithm returns *last*.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng1*>> **find_end**(*ExPolicy* &&policy, *Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns the last subsequence of elements *rng2* found in the range *rng* using the given predicate *f* to compare elements.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(begin(rng2), end(rng2)) and *N* = distance(begin(rng), end(rng)).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it

executes the assignments.

- **Rng1** – The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Rng2** – The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range of type dereferenced *iterator_t<Rng1>* as a projection operation before the function *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range of type dereferenced *iterator_t<Rng2>* as a projection operation before the function *op* is invoked.

Returns The *find_end* algorithm returns a *hpx::future<iterator_t<Rng> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *iterator_t<Rng>* otherwise. The *find_end* algorithm returns an iterator to the beginning of the last subsequence *rng2* in range *rng*. If the length of the subsequence *rng2* is greater than the length of the range *rng*, *end(rng)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng)* is also returned.

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter1*> **find_end**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns the last subsequence of elements [first2, last2) found in the range [first1, last1) using the given predicate *f* to compare elements.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in

unspecified threads, and indeterminately sequenced within each thread.

This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range of type dereferenced *iterator_t<Rng1>* as a projection operation before the function *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range of type dereferenced *iterator_t<Rng2>* as a projection operation before the function *op* is invoked.

Returns The *find_end* algorithm returns a *hpx::future<iterator_t<Rng> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *iterator_t<Rng>* otherwise. The *find_end* algorithm returns an iterator to the beginning of the last subsequence *rng2* in range *rng*. If the length of the subsequence *rng2* is greater than the length of the range *rng*, *end(rng)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng)* is also returned.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng1*> **find_end**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns the last subsequence of elements *rng2* found in the range *rng* using the given predicate *f* to compare elements.

This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(begin(rng2), end(rng2)) and *N* = distance(begin(rng), end(rng)).

Template Parameters

- **Rng1** – The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Rng2** – The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range of type dereferenced *iterator_t<Rng1>* as a projection operation before the function *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range of type dereferenced *iterator_t<Rng2>* as a projection operation before the function *op* is invoked.

Returns The *find_end* algorithm returns an iterator to the beginning of the last subsequence *rng2* in range *rng*. If the length of the subsequence *rng2* is greater than the length of the range *rng*, *end(rng)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng)* is also returned.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

Iter1 **find_end**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns the last subsequence of elements [first2, last2) found in the range [first1, last1) using the given predicate *f* to compare elements.

This overload of *find_end* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most S*(N-S+1) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **Iter1** – The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range of type dereferenced *iterator_t<Rng1>* as a projection

operation before the function *op* is invoked.

- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range of type dereferenced *iterator_t<Rng2>* as a projection operation before the function *op* is invoked.

Returns The *find_end* algorithm returns an iterator to the beginning of the last subsequence *rng2* in range *rng*. If the length of the subsequence *rng2* is greater than the length of the range *rng*, *end(rng)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng)* is also returned.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng1*>> **find_first_of**(*ExPolicy* &&policy, *Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range *rng1* for any elements in the range *rng2*. Uses binary predicate *p* to compare elements

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (S*N) comparisons where *S* = distance(begin(rng2), end(rng2)) and *N* = distance(begin(rng1), end(rng1)).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.

- **Rng2** – The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng1>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng1>* before the function *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng2>* before the function *op* is invoked.

Returns The *find_end* algorithm returns a *hpx::future<iterator_t<Rng1> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *iterator_t<Rng1>* otherwise. The *find_first_of* algorithm returns an iterator to the first element in the range *rng1* that is equal to an element from the range *rng2*. If the length of the subsequence *rng2* is greater than the length of the range *rng1*, *end(rng1)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng1)* is also returned.

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter1*> **find_first_of**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first1, last1) for any elements in the range [first2, last2). Uses binary predicate *p* to compare elements

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *find_first_of* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (S*N) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng1>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng1>* before the function *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng2>* before the function *op* is invoked.

Returns The *find_end* algorithm returns a *hpx::future<iterator_t<Rng1> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *iterator_t<Rng1>* otherwise. The *find_first_of* algorithm returns an iterator to the first element in the range *rng1* that is equal to an element from the range *rng2*. If the length of the subsequence *rng2* is greater than the length of the range *rng1*, *end(rng1)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng1)* is also returned.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng1*> **find_first_of**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range *rng1* for any elements in the range *rng2*. Uses binary predicate *p* to compare elements

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (S*N) comparisons where *S* = distance(begin(rng2), end(rng2)) and *N* = distance(begin(rng1), end(rng1)).

Template Parameters

- **Rng1** – The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Rng2** – The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng2*.

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng1>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng1>* before the function *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng2>* before the function *op* is invoked.

Returns The *find_first_of* algorithm returns an iterator to the first element in the range *rng1* that is equal to an element from the range *rng2*. If the length of the subsequence *rng2* is greater than the length of the range *rng1*, *end(rng1)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng1)* is also returned.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

Iter1 **find_first_of**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first1, last1) for any elements in the range [first2, last2). Uses binary predicate *p* to compare elements

This overload of *find_first_of* is available if the user decides to provide the algorithm their own predicate *op*.

Note: Complexity: at most (S*N) comparisons where *S* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- **Iter1** – The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements in *rng2*.

Parameters

- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng1>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng1>* before the function *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *iterator_t<Rng2>* before the function *op* is invoked.

Returns The *find_first_of* algorithm returns an iterator to the first element in the range *rng1* that is equal to an element from the range *rng2*. If the length of the subsequence *rng2* is greater than the length of the range *rng1*, *end(rng1)* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *end(rng1)* is also returned.

hpx::ranges::for_each, hpx::ranges::for_each_n

Defined in header hpx/algorithm.hpp[627].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
for_each_result<*InIter*, *F*> **for_each**(*InIter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Applies *f* to the result of dereferencing every iterator in the range [first, last).

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Note: Complexity: Applies *f* exactly *last* - *first* times.

Template Parameters

- **InIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

[627] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Returns {last, HPX_MOVE(f)} where last is the iterator corresponding to the input sentinel last.

template<typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
for_each_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *F*> **for_each**(*Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Applies *f* to the result of dereferencing every iterator in the given range *rng*.

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Note: Complexity: Applies *f* exactly *size(rng)* times.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns {std::end(rng), HPX_MOVE(f)}

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **for_each**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *F* &&f, *Proj* &&proj = *Proj*())

Applies *f* to the result of dereferencing every iterator in the range [first, last).

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Applies *f* exactly *last* - *first* times.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *for_each* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **for_each**(*ExPolicy* &&policy, *Rng* &&rng, *F* &&f, *Proj* &&proj = *Proj*())

Applies *f* to the result of dereferencing every iterator in the given range *rng*.

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Applies *f* exactly *size(rng)* times.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  <ignored> pred(const Type &a);
  ```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *for_each* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

template<typename **InIter**, typename **Size**, typename **F**, typename **Proj** = *hpx*::identity>
for_each_n_result<*InIter*, *F*> **for_each_n**(*InIter* first, *Size* count, *F* &&f, *Proj* &&proj = *Proj*())

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

Note: Complexity: Applies *f* exactly *count* times.

Template Parameters

- **InIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns It returns *last*.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **F**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **for_each_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *F* &&f, *Proj* &&proj = *Proj*())

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

If *f* returns a result, the result is ignored.

If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply non-constant functions through the dereferenced iterator.

Unlike its sequential form, the parallel overload of *for_each* does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Applies *f* exactly *count* times.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *for_each* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

hpx::ranges::experimental::for_loop, hpx::ranges::experimental::for_loop_strided, hpx::ranges::experimental::for_loop_n, hpx::ranges::experimental::for_loop_n_strided

Defined in header hpx/algorithm.hpp[628].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

namespace **experimental**

[628] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*>::type **for_loop**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *Args*&&... args)

The for_loop implements loop functionality over a range specified by iterator bounds. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: *Iter* shall meet the requirements of a forward iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter** – The type of the iteration variable (forward iterator).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for Iter.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Iter const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

template<typename **Iter**, typename **Sent**, typename ...**Args**>
void **for_loop**(*Iter* first, *Sent* last, *Args*&&... args)

The for_loop implements loop functionality over a range specified by iterator bounds. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *Iter* shall meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **Iter** – The type of the iteration variable (input iterator).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for Iter.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Iter const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

template<typename **ExPolicy**, typename **R**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*>::type **for_loop**(*ExPolicy* &&policy, *R* &&rng, *Args*&&... args)

The for_loop implements loop functionality over a range specified by a range. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: *Rng::iterator* shall meet the requirements of a forward iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **R** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to theof the sequence of elements the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Rng::iterator const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

template<typename **Rng**, typename ...**Args**>
void **for_loop**(*Rng* &&rng, *Args*&&... args)

The for_loop implements loop functionality over a range specified by a range. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *Rng::iterator* shall meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **rng** – Refers to theof the sequence of elements the algorithm will be applied to.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Rng::iterator const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **S**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **for_loop_strided**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *S* stride, *Args*&&... args)

The for_loop_strided implements loop functionality over a range specified by iterator bounds. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: *Iter* shall meet the requirements of a forward iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **`Iter`** – The type of the iteration variable (forward iterator).
- **`Sent`** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for Iter.
- **`S`** – The type of the stride variable. This should be an integral type.
- **`Args`** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **`last`** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **`stride`** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if Iter meets the requirements a bidirectional iterator.
- **`args`** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Iter const& a, ...);
```

The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop_strided* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

template<typename **Iter**, typename **Sent**, typename **S**, typename ...**Args**>
void **for_loop_strided**(*Iter* first, *Sent* last, *S* stride, *Args*&&... args)

The for_loop_strided implements loop functionality over a range specified by iterator bounds. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop_strided without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *Iter* shall meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **Iter** – The type of the iteration variable (input iterator).
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for Iter.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **stride** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if Iter meets the requirements a bidirectional iterator.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Iter const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

template<typename **ExPolicy**, typename **Rng**, typename **S**, typename ...**Args**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*> **for_loop_strided**(*ExPolicy* &&policy, *Rng* &&rng, *S* stride, *Args*&&... args)

The for_loop_strided implements loop functionality over a range specified by a range. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

Requires: *Rng::iterator* shall meet the requirements of a forward iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to theof the sequence of elements the algorithm will be applied to.
- **stride** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if Rng::iterator meets the requirements a bidirectional iterator.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

```
<ignored> pred(Rng::iterator const& a, ...);
```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

Returns The *for_loop_strided* algorithm returns a *hpx::future<void>* if the execution policy is of type *hpx::execution::sequenced_task_policy* or *hpx::execution::parallel_task_policy* and returns *void* otherwise.

template<typename **Rng**, typename **S**, typename ...**Args**>
void **for_loop_strided**(*Rng* &&rng, *S* stride, *Args*&&... args)

The for_loop_strided implements loop functionality over a range specified by a range. These algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop_strided without specifying an execution policy is equivalent to specifying *hpx::execution::seq* as the execution policy.

Requires: *Rng::iterator* shall meet the requirements of an input iterator type. The *args* parameter pack shall have at least one element, comprising objects returned by invocations of *reduction* and/or *induction* function templates followed by exactly one element invocable element-access function, *f*. *f* shall meet the requirements of MoveConstructible.

Effects: Applies *f* to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the *args* parameter pack. The length of the input sequence is last - first.

The first element in the input sequence is specified by *first*. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the *args* parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

If the pack member is an object returned by a call to a reduction function listed in section, then the additional argument is a reference to a view of that reduction object. If the pack member is an object returned by a call to induction, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

Remarks: If *f* returns a result, the result is ignored.

Note: As described in the C++ standard, arithmetic on non-random-access iterators is performed using advance and distance.

Note: The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **S** – The type of the stride variable. This should be an integral type.
- **Args** – A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- **rng** – Refers to theof the sequence of elements the algorithm will be applied to.
- **stride** – Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if Rng::iterator meets the requirements a bidirectional iterator.
- **args** – The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

  ```
  <ignored> pred(Rng::iterator const& a, ...);
  ```

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

hpx::ranges::generate, hpx::ranges::generate_n

Defined in header hpx/algorithm.hpp[629].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[629] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Rng**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **generate**(*ExPolicy* &&policy, *Rng* &&rng, *F* &&f)

Assign each element in range [first, last) a value generated by the given function object f

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

  The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

Returns The *replace_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **generate**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *F* &&f)

Assign each element in range [first, last) a value generated by the given function object f

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *generate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

  The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

Returns The *replace_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

template<typename **Rng**, typename **F**>
hpx::*traits*::range_iterator_t<*Rng*> **generate**(*Rng* &&rng, *F* &&f)

Assign each element in range [first, last) a value generated by the given function object f

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

  The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

Returns The *replace_if* algorithm returns *last*.

template<typename **Iter**, typename **Sent**, typename **F**>
Iter **generate**(*Iter* first, *Sent* last, *F* &&f)

Assign each element in range [first, last) a value generated by the given function object f

Note: Complexity: Exactly *distance(first, last)* invocations of *f* and assignments.

Template Parameters

- **Iter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – generator function that will be called. signature of function should be equivalent to the following:

```
Ret fun();
```

The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.

Returns The *replace_if* algorithm returns *last*.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **generate_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *F* &&f)

Assigns each element in range [first, first+count) a value generated by the given function object g.

The assignments in the parallel *generate_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *generate_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *count* invocations of *f* and assignments, for count > 0.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements in the sequence the algorithm will be applied to.
- **f** – Refers to the generator function object that will be called. The signature of the function should be equivalent to

```
Ret fun();
```

The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *replace_if* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. It returns *last*.

template<typename **FwdIter**, typename **Size**, typename **F**>
FwdIter **generate_n**(*FwdIter* first, *Size* count, *F* &&f)

Assigns each element in range [first, first+count) a value generated by the given function object g.

Note: Complexity: Exactly *count* invocations of *f* and assignments, for count > 0.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements in the sequence the algorithm will be applied to.
- **f** – Refers to the generator function object that will be called. The signature of the function should be equivalent to

```
Ret fun();
```

The type *Ret* must be such that an object of type *OutputIt* can be dereferenced and assigned a value of type *Ret*.

Returns The *replace_if* algorithm returns *last*.

hpx::ranges::includes

Defined in header hpx/algorithm.hpp[630].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[630] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **includes**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: At most 2*(N1+N2-1) comparisons, where N1 = std::distance(first1, last1) and N2 = std::distance(first2, last2).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *includes* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.

- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *includes* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *includes* algorithm returns true every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **includes**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*.

Note: At most 2*(N1+N2-1) comparisons, where N1 = std::distance(first1, last1) and N2 = std::distance(first2, last2).

Template Parameters

- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *includes* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm

will be applied to.

- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *includes* algorithm returns true every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **includes**(*ExPolicy* &&policy, *Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *includes* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: At most 2*(N1+N2-1) comparisons, where N1 = std::distance(first1, last1) and N2 = std::distance(first2, last2).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *includes* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *includes* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *includes* algorithm returns true every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **includes**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty. The version expects both ranges to be sorted with the user supplied binary predicate *f*.

Note: At most 2*(N1+N2-1) comparisons, where N1 = std::distance(first1, last1) and N2 = std::distance(first2, last2).

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *includes* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This

defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as includes. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *includes* algorithm returns true every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty.

hpx::ranges::inclusive_scan

Defined in header hpx/algorithm.hpp[631].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **Op** = *std*::plus<typename *std*::iterator_traits<*InIter*>::value_type>>
inclusive_scan_result<*InIter*, *OutIter*> **inclusive_scan**(*InIter* first, *Sent* last, *OutIter* dest, *Op* &&op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

[631] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:
- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Op** – The type of the binary function object used for the reduction operation.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *inclusive_scan* algorithm returns *util::in_out_result<InIter, OutIter>*. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **Op** = *std*::plus<typename *std*::iterator_traits<*FwdIter1*>::value_type>>
parallel::*util*::detail::algorithm_result<*ExPolicy*, inclusive_scan_result<*FwdIter1*, *FwdIter2*>>::type **inclusive_scan**(*Exl*
&&
icy
Fw
firs
Ser
last
Fw
des
Op
&&

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:
- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Op** – The type of the binary function object used for the reduction operation.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *inclusive_scan* algorithm returns a *hpx::future<util::in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result<FwdIter1, FwdIter2>* otherwise. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **O**, typename **Op** = *std*::plus<typename *hpx*::*traits*::range_traits<*Rng*>::value_type>>

inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*> **inclusive_scan**(*Rng* &&rng, *O* dest, *Op* &&op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:

- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Op** – The type of the binary function object used for the reduction operation.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *inclusive_scan* algorithm returns *util::in_out_result<traits::range_iterator_t<Rng>*, O> The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **O**, typename **Op** = *std*::plus<typename *hpx*::*traits*::range_traits<*Rng*>::value_type>>

parallel::*util*::detail::algorithm_result<*ExPolicy*, inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*>> **inclusive**

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, …, aN) is defined as:
- a1 when N is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Op** – The type of the binary function object used for the reduction operation.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

Returns The *inclusive_scan* algorithm returns a *hpx::future<util::in_out_result* <traits::range_iterator_t<Rng>, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result* <traits::range_iterator_t<Rng>, O> otherwise. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **Op**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type>
inclusive_scan_result<*InIter*, *OutIter*> **inclusive_scan**(*InIter* first, *Sent* last, *OutIter* dest, *Op* &&op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters
- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Op** – The type of the binary function object used for the reduction operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- **init** – The initial value for the generalized sum.

Returns The *inclusive_scan* algorithm returns *util::in_out_result<InIter, OutIter>*. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **Op**, typename **T** = typename *std*::iterator_traits<*FwdIter1*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, inclusive_scan_result<*FwdIter1*, *FwdIter2*>>::type **inclusive_scan**(*Ex*
&&
icy
InI
firs
Ser
last
Ou
Iter
des
Op
&&
T
init

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Op** – The type of the binary function object used for the reduction operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **init** – The initial value for the generalized sum.

Returns The *inclusive_scan* algorithm returns a *hpx::future<util::in_out_result<InIter, OutIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result<InIter, OutIter>* otherwise. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **O**, typename **Op**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*> **inclusive_scan**(*Rng* &&rng, *O* dest, *Op* &&op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1

- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Op** – The type of the binary function object used for the reduction operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **init** – The initial value for the generalized sum.

Returns The *inclusive_scan* algorithm returns *util::in_out_result<traits::range_iterator_t<Rng>*, O> The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **O**, typename **Op**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*>> **inclusive**

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, …, *(first + (i - result))).

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *exclusive_scan* and *inclusive_scan* is that *inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *inclusive_scan* may be non-deterministic.

Note: Complexity: O(*last* - *first*) applications of the predicate *op*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN)) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Op** – The type of the binary function object used for the reduction operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **init** – The initial value for the generalized sum.

Returns The *inclusive_scan* algorithm returns a *hpx::future<util::in_out_result* <traits::range_iterator_t<Rng>, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *util::in_out_result* <traits::range_iterator_t<Rng>, O> otherwise. The *inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied

hpx::ranges::is_heap, hpx::ranges::is_heap_until

Defined in header hpx/algorithm.hpp[632].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[632] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Rng**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_heap**(*ExPolicy* &&policy, *Rng* &&rng, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_heap**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

template<typename **Rng**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
bool **is_heap**(*Rng* &&rng, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap* algorithm returns *bool*. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

template<typename **Iter**, typename **Sent**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
bool **is_heap**(*Iter* first, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap* algorithm returns *bool*. The *is_heap* algorithm returns whether the range is max heap. That is, true if the range is max heap, false otherwise.

template<typename **ExPolicy**, typename **Rng**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **is_heap_until**(*ExPolicy* &&policy, *Rng* &&rng, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap_until* algorithm returns a *hpx::future<RandIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandIter* otherwise. The *is_heap_until* algorithm returns the upper bound of the largest range beginning at first which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>

hpx::parallel::util::detail::algorithm_result_t<ExPolicy, Iter> **is_heap_until**(ExPolicy &&policy, Iter first, Sent last, Comp &&comp = Comp(), Proj &&proj = Proj())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap_until* algorithm returns a *hpx::future<RandIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandIter* otherwise. The *is_heap_until* algorithm returns the upper bound of the largest range beginning at first which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap.

template<typename **Rng**, typename **Comp** = hpx::parallel::detail::less, typename **Proj** = hpx::identity>
hpx::traits::range_iterator_t<Rng> **is_heap_until**(Rng &&rng, Comp &&comp = Comp(), Proj &&proj = Proj())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap_until* algorithm returns *RandIter*. The *is_heap_until* algorithm returns the upper bound of the largest range beginning at first which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap.

template<typename **Iter**, typename **Sent**, typename **Comp** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
Iter **is_heap_until**(*Iter* first, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note: Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – *comp* is a callable object. The return value of the INVOKE operation applied to an object of type *Comp*, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_heap_until* algorithm returns *RandIter*. The *is_heap_until* algorithm returns the upper bound of the largest range beginning at first which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap.

hpx::ranges::is_partitioned

Defined in header hpx/algorithm.hpp[633].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
bool **is_partitioned**(*FwdIter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Determines if the range [first, last) is partitioned.

Note: Complexity: at most (N) predicate evaluations where *N* = distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_partitioned* algorithm returns *bool*. The *is_partitioned* algorithm returns true if each element in the sequence for which pred returns true precedes those for which pred returns false. Otherwise is_partitioned returns false. If the range [first, last) contains less than two elements, the function is always true.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_partitioned**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

[633] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Determines if the range [first, last) is partitioned.

The predicate operations in the parallel *is_partitioned* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_partitioned* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N) predicate evaluations where *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of the function/function object to use (deduced). *Pred* must be *Copy-Constructible* when using a parallel policy.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_partitioned* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_partitioned* algorithm returns true if each element in the sequence for which pred returns true precedes those for which pred returns false. Otherwise is_partitioned returns false. If the range [first, last) contains less than two elements, the function is always true.

template<typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
bool **is_partitioned**(*Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Determines if the range rng is partitioned.

Note: Complexity: at most (N) predicate evaluations where *N* = std::size(rng).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

```
bool pred(const Type &a);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_partitioned* algorithm returns *bool*. The *is_partitioned* algorithm returns true if each element in the sequence for which pred returns true precedes those for which pred returns false. Otherwise is_partitioned returns false. If the range rng contains less than two elements, the function is always true.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_partitioned**(*ExPolicy* &&policy, *Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Determines if the range [first, last) is partitioned.

The predicate operations in the parallel *is_partitioned* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_partitioned* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N) predicate evaluations where *N* = std::size(rng).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). *Pred* must be *Copy-Constructible* when using a parallel policy.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Refers to the unary predicate which returns true for elements expected to be found in the beginning of the range. The signature of the function should be equivalent to

```
bool pred(const Type &a);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_partitioned* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_partitioned* algorithm returns true if each element in the sequence for which pred returns true precedes those for which pred returns false. Otherwise is_partitioned returns false. If the range rng contains less than two elements, the function is always true.

hpx::ranges::is_sorted, hpx::ranges::is_sorted_until

Defined in header hpx/algorithm.hpp[634].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
bool **is_sorted**(*FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Determines if the range [first, last) is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm executes in sequential order in the calling thread.

Note: Complexity: at most (N+S-1) comparisons where N = distance(first, last). S = number of partitions

Template Parameters

- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of an optional function/function object to use.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.

[634] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted* algorithm returns a *bool*. The *is_sorted* algorithm returns true if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_sorted**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Determines if the range [first, last) is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *Copy-Constructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.

- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_sorted* algorithm returns a bool if each element in the sequence [first, last) satisfies the predicate passed. If the range [first, last) contains less than two elements, the function always returns true.

template<typename **Rng**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
bool **is_sorted**(*Rng* &&rng, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Determines if the range rng is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm executes in sequential order in the calling thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = size(rng). *S* = number of partitions

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted* algorithm returns a *bool*. The *is_sorted* algorithm returns true if each element in the rng satisfies the predicate passed. If the range rng contains less than two elements, the function always returns true.

template<typename **ExPolicy**, typename **Rng**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **is_sorted**(*ExPolicy* &&policy, *Rng* &&rng, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Determines if the range rng is sorted. Uses pred to compare elements.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_sorted* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = size(rng). *S* = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *Copy-Constructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

  ```
  bool pred(const Type &a, const Type &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted* algorithm returns a *hpx::future<bool>* if the execution policy is of type *task_execution_policy* and returns *bool* otherwise. The *is_sorted* algorithm returns a bool if each element in the range rng satisfies the predicate passed. If the range rng contains less than two elements, the function always returns true.

template<typename **FwdIter**, typename **Sent**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
FwdIter **is_sorted_until**(*FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of an optional function/function object to use.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

  ```
  bool pred(const Type &a, const Type &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted_until* algorithm returns a *FwdIter*. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, last is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **is_sorted_until**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Returns the first element in the range [first, last) that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = distance(first, last). *S* = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted_until* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of that the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

  ```
  bool pred(const Type &a, const Type &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted_until* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, last is returned.

template<typename **Rng**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **is_sorted_until**(*Rng* &&rng, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Returns the first element in the range rng that is not sorted. Uses a predicate to compare elements or the less than operator.

Note: Complexity: at most (N+S-1) comparisons where *N* = size(rng). *S* = number of partitions

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted_until* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be

equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted_until* returns *FwdIter*. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, last is returned.

template<typename **ExPolicy**, typename **Rng**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **is_sorted_until**(*ExPolicy* &&policy, *Rng* &&rng, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Returns the first element in the range rng that is not sorted. Uses a predicate to compare elements or the less than operator.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *is_sorted_until* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (N+S-1) comparisons where *N* = size(rng). *S* = number of partitions

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted_until* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.

- **pred** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *is_sorted_until* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *is_sorted_until* algorithm returns the first unsorted element. If the sequence has less than two elements or the sequence is sorted, last is returned.

hpx::ranges::lexicographical_compare

Defined in header hpx/algorithm.hpp[635].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter1**, typename **Sent1**, typename **InIter2**, typename **Sent2**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity, typename **Pred** = *hpx*::*parallel*::detail::less>
bool **lexicographical_compare**(*InIter1* first1, *Sent1* last1, *InIter2* first2, *Sent2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: At most 2 * min(N1, N2) applications of the comparison operation, where N1 = std::distance(first1, last) and N2 = std::distance(first2, last2).

Note: Lexicographical comparison is an operation with the following properties
- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other

[635] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Template Parameters

- **InIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter1.
- **InIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function for FwdIter1. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for FwdIter2. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **pred** – Refers to the comparison function that the first and second ranges will be applied to
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *lexicographically_compare* algorithm returns *bool*. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity, typename **Pred** = *hpx*::*parallel*::detail::less>

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **lexicographical_compare**(*ExPolicy* &&policy, *FwdIter1* first1, *Sent1* last1, *FwdIter2* first2, *Sent2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2 * min(N1, N2) applications of the comparison operation, where N1 = std::distance(first1, last) and N2 = std::distance(first2, last2).

Note: Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter1.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter2.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function for FwdIter1. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for FwdIter2. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **pred** – Refers to the comparison function that the first and second ranges will be applied to
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *lexicographically_compare* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

template<typename **Rng1**, typename **Rng2**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity, typename **Pred** = *hpx*::*parallel*::detail::less>
bool **lexicographical_compare**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks if the first range rng1 is lexicographically less than the second range rng2. uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: At most 2 * min(N1, N2) applications of the comparison operation, where N1 = std::distance(std::begin(rng1), std::end(rng1)) and N2 = std::distance(std::begin(rng2), std::end(rng2)).

Note: Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other

- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function for elements of the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for elements of the second range. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Refers to the comparison function that the first and second ranges will be applied to
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *lexicographically_compare* algorithm returns *bool*. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity, typename **Pred** = *hpx*::*parallel*::detail::less>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **lexicographical_compare**(*ExPolicy* &&policy, *Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks if the first range rng1 is lexicographically less than the second range rng2. uses a provided predicate to compare elements.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *lexicographical_compare* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2 * min(N1, N2) applications of the comparison operation, where N1 = std::distance(std::begin(rng1), std::end(rng1)) and N2 = std::distance(std::begin(rng2), std::end(rng2)).

Note: Lexicographical comparison is an operation with the following properties

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*
- An empty range is lexicographically *less* than any non-empty range
- Two empty ranges are lexicographically *equal*

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`Rng1`** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **`Rng2`** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **`Pred`** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *lexicographical_compare* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **`Proj1`** – The type of an optional projection function for elements of the first range. This defaults to *hpx::identity*
- **`Proj2`** – The type of an optional projection function for elements of the second range. This defaults to *hpx::identity*

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`rng1`** – Refers to the sequence of elements the algorithm will be applied to.
- **`rng2`** – Refers to the sequence of elements the algorithm will be applied to.
- **`pred`** – Refers to the comparison function that the first and second ranges will be applied to
- **`proj1`** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **`proj2`** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *lexicographically_compare* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *lexicographically_compare* algorithm returns true if the first range is lexicographically less, otherwise it returns false. range [first2, last2), it returns false.

hpx::ranges::make_heap

Defined in header hpx/algorithm.hpp[636].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Comp**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **make_heap**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *Comp* &&comp, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

```
bool comp(const Type &a, const Type &b);
```

[636] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *RndIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Comp**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **make_heap**(*ExPolicy* &&policy, *Rng* &&rng, *Comp* &&comp, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

```
bool comp(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *RndIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **make_heap**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **make_heap**(*ExPolicy* &&policy, *Rng* &&rng, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

The predicate operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *sequential_execution_policy* executes in sequential order in the calling thread.

The comparison operations in the parallel *make_heap* algorithm invoked with an execution policy object of type *parallel_execution_policy* or *parallel_task_execution_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. It returns *last*.

template<typename **Iter**, typename **Sent**, typename **Comp**, typename **Proj** = *hpx*::identity>
Iter **make_heap**(*Iter* first, *Sent* last, *Comp* &&comp, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **comp** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

```
bool comp(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *RndIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

template<typename **Rng**, typename **Comp**, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **make_heap**(*Rng* &&rng, *Comp* &&comp, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – Refers to the binary predicate which returns true if the first argument should be treated as less than the second. The signature of the function should be equivalent to

```
bool comp(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *RndIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

template<typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity>
Iter **make_heap**(*Iter* first, *Sent* last, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **Iter** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

template<typename **Rng**, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **make_heap**(*Rng* &&rng, *Proj* &&proj = *Proj*{})

Constructs a *max heap* in the range [first, last).

Note: Complexity: at most (3*N) comparisons where *N* = distance(first, last).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *make_heap* algorithm returns *Iter*. It returns *last*.

hpx::ranges::merge, hpx::ranges::inplace_merge

Defined in header hpx/algorithm.hpp[637].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*ranges*::merge_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

[637] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs O(std::distance(first1, last1) + std::distance(first2, last2)) applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function to be used for elements of the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first range of elements the algorithm will be applied to.
- **rng2** – Refers to the second range of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

  ```
  bool comp(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter1* and *Iter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual comparison *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual comparison *comp* is invoked.

Returns The *merge* algorithm returns a *hpx::future<merge_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *merge_result<Iter1, Iter2, Iter3>* otherwise. The *merge* algorithm returns the tuple of the source iterator *last1*, the source iterator *last2*, the destination iterator to the end of the *dest* range.

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*ranges*::merge_result<*Iter1*, *Iter2*, *Iter3*>>::type **merge**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Comp* &&comp = *Comp*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs O(std::distance(first1, last1) + std::distance(first2, last2)) applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an random access iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.

- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an random access iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function to be used for elements of the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter1* and *Iter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual comparison *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual comparison *comp* is invoked.

Returns The *merge* algorithm returns a *hpx::future<merge_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *merge_result<Iter1, Iter2, Iter3>* otherwise. The *merge* algorithm returns the tuple of the source iterator *last1*, the source iterator *last2*, the destination iterator to the end of the *dest* range.

template<typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::*ranges*::merge_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range_iterator_t<*Rng2*>, *Iter3*> **merge**(*Rng1* &&rng1, *Rng2* &&rng2, *Iter3* dest, *Comp* &&comp = *Comp*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

Note: Complexity: Performs O(std::distance(first1, last1) + std::distance(first2, last2)) applications of the comparison *comp* and the each projection.

Template Parameters

- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function to be used for elements of the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second range. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first range of elements the algorithm will be applied to.
- **rng2** – Refers to the second range of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter1* and *Iter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the

elements of the first range as a projection operation before the actual comparison *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual comparison *comp* is invoked.

Returns The *merge* algorithm returns *merge_result<Iter1, Iter2, Iter3>*. The *merge* algorithm returns the tuple of the source iterator *last1*, the source iterator *last2*, the destination iterator to the end of the *dest* range.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*ranges*::merge_result<*Iter1*, *Iter2*, *Iter3*> **merge**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Comp* &&comp = *Comp*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

Note: Complexity: Performs O(std::distance(first1, last1) + std::distance(first2, last2)) applications of the comparison *comp* and the each projection.

Template Parameters
- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an random access iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an random access iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function to be used for elements of the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second range. This defaults to *hpx::identity*

Parameters
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter1* and *Iter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual comparison *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual comparison *comp* is invoked.

Returns The *merge* algorithm returns *merge_result<Iter1, Iter2, Iter3>*. The *merge* algorithm returns the tuple of the source iterator *last1*, the source iterator *last2*, the destination iterator to the end of the *dest* range.

template<typename **ExPolicy**, typename **Rng**, typename **Iter**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **inplace_merge**(*ExPolicy* &&policy, *Rng* &&rng, *Iter* middle, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the range of elements the algorithm will be applied to.

- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *inplace_merge* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. The *inplace_merge* algorithm returns the source iterator *last*

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **inplace_merge**(*ExPolicy* &&policy, *Iter* first, *Iter* middle, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *inplace_merge* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.

- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *inplace_merge* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. The *inplace_merge* algorithm returns the source iterator *last*

template<typename **Rng**, typename **Iter**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj** = *hpx*::identity>
Iter **inplace_merge**(*Rng* &&rng, *Iter* middle, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the range of elements the algorithm will be applied to.
- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *inplace_merge* algorithm returns *Iter*. The *inplace_merge* algorithm returns the source iterator *last* template<typename **Iter**, typename **Sent**, typename **Comp** = *hpx*::*ranges*::less, typename **Proj** = *hpx*::identity>

Iter **inplace_merge**(*Iter* first, *Iter* middle, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

Note: Complexity: Performs O(std::distance(first, last)) applications of the comparison *comp* and the each projection.

Template Parameters

- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Comp** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *inplace_merge* requires *Comp* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **middle** – Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **comp** – *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

```
bool comp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *Iter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *inplace_merge* algorithm *Iter*. The *inplace_merge* algorithm returns the source iterator *last*

hpx::ranges::min_element, hpx::ranges::max_element, hpx::ranges::minmax_element

Defined in header hpx/algorithm.hpp[638].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[638] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **FwdIter**, typename **Sent**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
FwdIter **min_element**(*FwdIter* first, *Sent* last, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **F** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *min_element* algorithm returns *FwdIter*. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **Rng**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **min_element**(*Rng* &&rng, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

- **F** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *min_element* algorithm returns a *hpx::traits::range_iterator<Rng>::type* otherwise. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **F** = hpx::parallel::detail::less, typename **Proj** = hpx::identity>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, FwdIter> **min_element**(ExPolicy &&policy, FwdIter first, Sent last, F &&f = F(), Proj &&proj = Proj())

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *min_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

- **f** – The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *min_element* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **ExPolicy**, typename **Rng**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **min_element**(*ExPolicy* &&policy, *Rng* &&rng, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the smallest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *min_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *min_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *min_element* algorithm returns a *hpx::future<hpx::traits::range_iterator<Rng>::type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *min_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **FwdIter**, typename **Sent**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
FwdIter **max_element**(*FwdIter* first, *Sent* last, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *max_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **F** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *max_element* algorithm returns a *FwdIter*. The *max_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **Rng**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **max_element**(*Rng* &&rng, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *max_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *max_element* algorithm returns a *hpx::traits::range_iterator<Rng>::type* otherwise. The *max_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **max_element**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *max_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *max_element* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *max_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **ExPolicy**, typename **Rng**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **max_element**(*ExPolicy* &&policy, *Rng* &&rng, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly *max*(N-1, 0) comparisons, where N = std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *max_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *max_element* algorithm returns a *hpx::future<hpx::traits::range_iterator<Rng>::type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *max_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **FwdIter**, typename **Sent**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
minmax_element_result<*FwdIter*> **minmax_element**(*FwdIter* first, *Sent* last, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The assignments in the parallel *minmax_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: At most *max*(floor(3/2*(N-1)), 0) applications of the predicate, where N = std::distance(first, last).

Template Parameters

- **FwdIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.

- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **F** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. This argument is optional and defaults to std::less. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *minmax_element* algorithm returns a *minmax_element_result<FwdIter, FwdIter>* The *minmax_element* algorithm returns a min_max_result consisting of an iterator to the smallest element as the min element and an iterator to the greatest element as the max element. Returns minmax_element_result{first, first} if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

template<typename **Rng**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
minmax_element_result<*hpx*::*traits*::range_iterator_t<*Rng*>> **minmax_element**(*Rng* &&rng, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The assignments in the parallel *minmax_element* algorithm execute in sequential order in the calling thread.

Note: Complexity: At most *max*(floor(3/2*(N-1)), 0) applications of the predicate, where N = std::distance(first, last).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. This argument is optional and defaults to std::less. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *minmax_element* algorithm returns a *minmax_element_result<hpx::traits::range_iterator<Rng>::type* , hpx::traits::range_iterator<Rng>::type> The *minmax_element* algorithm returns a min_max_result consisting of an range iterator to the smallest element as the min element and an range iterator to the greatest element as the max element. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, minmax_element_result<*FwdIter*>> **minmax_element**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *F* &&f = *F*(), *Proj* &&proj = *Proj*())

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *max*(floor(3/2*(N-1)), 0) applications of the predicate, where N = std::distance(first, last).

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form,

the parallel overload of *minmax_element* requires *F* to meet the requirements of *CopyConstructible*.

- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. This argument is optional and defaults to std::less. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *minmax_element* algorithm returns a *minmax_element_result<FwdIter, FwdIter>* The *minmax_element* algorithm returns a min_max_result consisting of an iterator to the smallest element as the min element and an iterator to the greatest element as the max element. Returns minmax_element_result{first, first} if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

template<typename **ExPolicy**, typename **Rng**, typename **F** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, minmax_element_result<*hpx*::*traits*::range_iterator_t<*Rng*>>> **min**

Finds the greatest element in the range [first, last) using the given comparison function *f*.

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparisons in the parallel *minmax_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *max*(floor(3/2*(N-1)), 0) applications of the predicate, where N =

std::distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *minmax_element* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – The binary predicate which returns true if the the left argument is less than the right element. This argument is optional and defaults to std::less. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *minmax_element* algorithm returns a *minmax_element_result<hpx::traits::range_iterator<Rng>::type*, hpx::traits::range_iterator<Rng>::type> The *minmax_element* algorithm returns a min_max_result consisting of an range iterator to the smallest element as the min element and an range iterator to the greatest element as the max element. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

hpx::ranges::mismatch

Defined in header hpx/algorithm.hpp[639].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[639] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, mismatch_result<*Iter1*, *Iter2*>>::type **mismatch**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if the range [first1, last1) is mismatch to the range [first2, last2), and false otherwise.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *f*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of the predicate *f* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatchs *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent1** – The type of the source iterators used for the end of the first range (deduced).
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the source iterators used for the end of the second range (deduced).
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *mismatch* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *mismatch* algorithm returns true if the elements in the two ranges are mismatch, otherwise it returns false. If the length of the range [first1, last1) does not mismatch the length of the range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, mismatch_result<typename *hpx*::*traits*::range_traits<*Rng1*>::iterator_

Returns std::pair with iterators to the first two non-equivalent elements.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *mismatch* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most *last1* - *first1* applications of the predicate *f*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *mismatch* algorithm returns a *hpx::future<std::pair<FwdIter1, FwdIter2> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1, FwdIter2>* otherwise. The *mismatch* algorithm returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2, last2).

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
mismatch_result<*Iter1*, *Iter2*> **mismatch**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Returns true if the range [first1, last1) is mismatch to the range [first2, last2), and false otherwise.

Note: Complexity: At most min(last1 - first1, last2 - first2) applications of the predicate *f*. If *FwdIter1* and *FwdIter2* meet the requirements of *RandomAccessIterator* and (last1 - first1) != (last2 - first2) then no applications of the predicate *f* are made.

Note: The two ranges are considered mismatch if, for every iterator i in the range [first1,last1), *i mismatchs *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

Template Parameters

- **Iter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the source iterators used for the end of the first range (deduced).
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the source iterators used for the end of the second range (deduced).
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.

- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *mismatch* algorithm returns *bool*. The *mismatch* algorithm returns true if the elements in the two ranges are mismatch, otherwise it returns false. If the length of the range [first1, last1) does not mismatch the length of the range [first2, last2), it returns false.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
mismatch_result<typename *hpx*::*traits*::range_traits<*Rng1*>::iterator_type, typename *hpx*::*traits*::range_traits<*Rng2*>::iterat

Returns std::pair with iterators to the first two non-equivalent elements.

Note: Complexity: At most *last1* - *first1* applications of the predicate *f*.

Template Parameters

- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function applied to the first range. This

defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second range. This defaults to *hpx::identity*

Parameters
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **op** – The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second range as a projection operation before the actual predicate *is* invoked.

Returns The *mismatch* algorithm returns *std::pair<FwdIter1, FwdIter2>*. The *mismatch* algorithm returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2, last2).

hpx::ranges::move

Defined in header hpx/algorithm.hpp[640].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, move_result<*Iter1*, *Iter2*>>::type **move**(*ExPolicy* &&policy, *Iter1* first, *Sent1* last, *Iter2* dest)

Moves the elements in the range *rng* to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

[640] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the source iterators used for the end of the first range (deduced).
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns a *hpx::future<ranges::move_result<iterator_t<Rng>*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::move_result<iterator_t<Rng>*, FwdIter2> otherwise. The *move* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element moved.

template<typename **ExPolicy**, typename **Rng**, typename **Iter2**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, move_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *Iter2*>>::type **move(**

Moves the elements in the range *rng* to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns a *hpx::future<ranges::move_result<iterator_t<Rng>*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::move_result<iterator_t<Rng>*, FwdIter2> otherwise. The *move* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element moved.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**>
move_result<*Iter1*, *Iter2*> **move**(*Iter1* first, *Sent1* last, *Iter2* dest)

Moves the elements in the range *rng* to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

Note: Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters
- **Iter1** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the source iterators used for the end of the first range (deduced).
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns *ranges::move_result<iterator_t<Rng>*, FwdIter2>. The *move* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element moved.

template<typename **Rng**, typename **Iter2**>
move_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *Iter2*> **move**(*Rng* &&rng, *Iter2* dest)

Moves the elements in the range *rng* to another range beginning at *dest*. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

Note: Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.

- **dest** – Refers to the beginning of the destination range.

Returns The *move* algorithm returns a *ranges::move_result<iterator_t<Rng>*, FwdIter2>. The *move* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element moved.

hpx::ranges::nth_element

Defined in header hpx/algorithm.hpp[641].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **RandomIt**, typename **Sent**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
RandomIt **nth_element**(*RandomIt* first, *RandomIt* nth, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by nth is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new nth element are less than or equal to the elements after the new nth element.

The comparison operations in the parallel *nth_element* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters
- **RandomIt** – The type of the source begin, nth, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Pred** – Comparison function object which returns true if the first argument is less than the second.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **nth** – Refers to the iterator defining the sort partition point
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

[641] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **pred** – Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. This defaults to hpx::identity.

Returns The *nth_element* algorithm returns returns *RandomIt*. The *nth_element* algorithm returns an iterator equal to last.

template<typename **ExPolicy**, typename **RandomIt**, typename **Sent**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *RandomIt*> **nth_element**(*ExPolicy* &&policy, *RandomIt* first, *RandomIt* nth, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by nth is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new nth element are less than or equal to the elements after the new nth element.

The comparison operations in the parallel *nth_element* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *nth_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandomIt** – The type of the source begin, nth, and end iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Pred** – Comparison function object which returns true if the first argument is less than the second.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **nth** – Refers to the iterator defining the sort partition point

- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred** – Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

  ```
  bool cmp(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. This defaults to hpx::identity.

Returns The *partition* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *parallel_task_policy* and returns *RandomIt* otherwise. The *nth_element* algorithm returns an iterator equal to last.

template<typename **Rng**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **nth_element**(*Rng* &&rng, *hpx*::*traits*::range_iterator_t<*Rng*> nth, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by nth is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new nth element are less than or equal to the elements after the new nth element.

The comparison operations in the parallel *nth_element* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Pred** – Comparison function object which returns true if the first argument is less than the second.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **nth** – Refers to the iterator defining the sort partition point
- **pred** – Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

  ```
  bool cmp(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. This defaults to hpx::identity.

Returns The *nth_element* algorithm returns returns *hpx::traits::range_iterator_t<Rng>*. The *nth_element* algorithm returns an iterator equal to last.

template<typename **ExPolicy**, typename **Rng**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **nth_element**(*ExPolicy* &&policy, *Rng* &&rng, *hpx*::*traits*::range_ite nth, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that the element pointed at by nth is changed to whatever element would occur in that position if [first, last) were sorted and all of the elements before this new nth element are less than or equal to the elements after the new nth element.

The comparison operations in the parallel *nth_element* invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *nth_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in std::distance(first, last) on average. O(N) applications of the predicate, and O(N log N) swaps, where N = last - first.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- **Pred** – Comparison function object which returns true if the first argument is less than the second.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **nth** – Refers to the iterator defining the sort partition point
- **pred** – Specifies the comparison function object which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of this comparison function should be equivalent to:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type must be such that an object of type *randomIt* can be dereferenced and then implicitly converted to Type. This defaults to std::less<>.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. This defaults to hpx::identity.

Returns The *partition* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>>* if the execution policy is of type *parallel_task_policy* and returns *hpx::traits::range_iterator_t<Rng>* otherwise. The *nth_element* algorithm returns an iterator equal to last.

hpx::ranges::partial_sort

Defined in header hpx/algorithm.hpp[642].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **RandomIt**, typename **Sent**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
RandomIt **partial_sort**(*RandomIt* first, *RandomIt* middle, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

The assignments in the parallel *partial_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Approximately (last - first) * log(middle - first) comparisons.

Template Parameters

- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

642 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the middle of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. Comp defaults to detail::less.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *partial_sort* algorithm returns *RandomIt*. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

template<typename **ExPolicy**, typename **RandomIt**, typename **Sent**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *RandomIt*>::type **partial_sort**(*ExPolicy* &&policy, *RandomIt* first, *RandomIt* middle, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Approximately (last - first) * log(middle - first) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

- **middle** – Refers to the middle of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. Comp defaults to detail::less.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *partial_sort* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

template<typename **Rng**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **partial_sort**(*Rng* &&rng, *hpx*::*traits*::range_iterator_t<*Rng*> middle, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Places the first middle - first elements from the range [first, last) as sorted with respect to comp into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

The assignments in the parallel *partial_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Approximately (last - first) * log(middle - first) comparisons.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the middle of the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. Comp defaults to detail::less.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *partial_sort* algorithm returns *hpx::traits::range_iterator_t<Rng>*. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>

parallel::*util*::detail::algorithm_result_t<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **partial_sort**(*ExPolicy* &&policy, *Rng* &&rng, *hpx*::*traits*::range_i mid-dle, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less;
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the middle of the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that

comp will not apply any non-constant function through the dereferenced iterator. Comp defaults to detail::less.

- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *partial_sort* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::traits::range_iterator_t<Rng>* otherwise. It returns *last*.

hpx::ranges::partial_sort_copy

Defined in header hpx/algorithm.hpp[643].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter**, typename **Sent1**, typename **RandIter**, typename **Sent2**, typename **Comp** = *ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
partial_sort_copy_result<*InIter*, *RandIter*> **partial_sort_copy**(*InIter* first, *Sent1* last, *RandIter* r_first, *Sent2* r_last, *Comp* &&comp = *Comp*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [r_first, r_last). At most r_last - r_first of the elements are placed sorted to the range [r_first, r_first + n) where n is the number of elements to sort (n = min(last - first, r_last - r_first)).

The assignments in the parallel *partial_sort_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(N log(min(D,N))), where N = std::distance(first, last) and D = std::distance(r_first, r_last) comparisons.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the source sentinel (deduced).This sentinel type must be a sentinel for InIter.
- **RandIter** – The type of the destination iterators used(deduced) This iterator type must meet the requirements of an random iterator.
- **Sent2** – The type of the destination sentinel (deduced).This sentinel type must be a sentinel for RandIter.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.

643 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **Proj1** – The type of an optional projection function for the input range. This defaults to *hpx::identity*.
- **Proj1** – The type of an optional projection function for the output range. This defaults to *hpx::identity*.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the sentinel value denoting the end of the sequence of elements the algorithm will be applied to.
- **r_first** – Refers to the beginning of the destination range.
- **r_last** – Refers to the sentinel denoting the end of the destination range.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to detail::less.
- **proj1** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate *comp* is invoked.

Returns The *partial_sort_copy* algorithm returns a returns *partial_sort_copy_result<InIter, RandIter>*. The algorithm returns {last, result_first + N}.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent1**, typename **RandIter**, typename **Sent2**, typename **Comp** = *ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, partial_sort_copy_result<*FwdIter*, *RandIter*>> **partial_sort_copy**(*E*
&
ic
F
fi
S
la
R
d
r_
S
r_
C
&
=
C
P
&
=
P
P
&
=
P

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [r_first, r_last). At most r_last - r_first of the elements are placed sorted to the range [r_first, r_first + n) where n is the number of elements to sort (n = min(last - first, r_last - r_first)).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(min(D,N))), where N = std::distance(first, last) and D = std::distance(r_first, r_last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the source sentinel (deduced).This sentinel type must be a sentinel for FwdIter.
- **RandIter** – The type of the destination iterators used(deduced) This iterator type must meet the requirements of an random iterator.
- **Sent2** – The type of the destination sentinel (deduced).This sentinel type must be a sentinel for RandIter.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.
- **Proj1** – The type of an optional projection function for the input range. This defaults to *hpx::identity*.
- **Proj1** – The type of an optional projection function for the output range. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the sentinel value denoting the end of the sequence of elements the algorithm will be applied to.
- **r_first** – Refers to the beginning of the destination range.
- **r_last** – Refers to the sentinel denoting the end of the destination range.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to detail::less.
- **proj1** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate *comp* is invoked.

Returns The *partial_sort_copy* algorithm returns a *hpx::future<partial_sort_copy_result<FwdIter, RandIter>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *partial_sort_copy_result<FwdIter, RandIter>* otherwise. The algorithm returns {last, result_first + N}.

template<typename **Rng1**, typename **Rng2**, typename **Comp** = *ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

partial_sort_copy_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range_iterator_t<*Rng2*>> **partial_sort_copy(**

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [r_first, r_last). At most r_last - r_first of the elements are placed sorted to the range [r_first, r_first + n) where n is the number of elements to sort (n = min(last - first, r_last - r_first)).

The assignments in the parallel *partial_sort_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(N log(min(D,N))), where N = std::distance(first, last) and D = std::distance(r_first, r_last) comparisons.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a input iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of a random iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.
- **Proj1** – The type of an optional projection function for the input range. This defaults to *hpx::identity*.
- **Proj2** – The type of an optional projection function for the output range. This defaults to *hpx::identity*.

Parameters

- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to detail::less.
- **proj1** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate *comp* is invoked.

Returns The *partial_sort_copy* algorithm returns *partial_sort_copy_result<range_iterator_t<Rng1>*, range_iterator_t<Rng2>>. The algorithm returns {last, result_first + N}.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Comp** = *ranges*::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
parallel::*util*::detail::algorithm_result_t<*ExPolicy*, partial_sort_copy_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*

Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [r_first, r_last). At most r_last - r_first of the elements are placed sorted to the range [r_first, r_first + n) where n is the number of elements to sort (n = min(last - first, r_last - r_first)).

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(min(D,N))), where N = std::distance(first, last) and D = std::distance(r_first, r_last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of a random iterator.
- **Comp** – The type of the function/function object to use (deduced). Comp defaults to detail::less.
- **Proj1** – The type of an optional projection function for the input range. This defaults to *hpx::identity*.
- **Proj2** – The type of an optional projection function for the output range. This defaults to *hpx::identity*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the source range.

- **rng2** – Refers to the destination range.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator. This defaults to detail::less.
- **proj1** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate *comp* is invoked.

Returns The *partial_sort_copy* algorithm returns a *hpx::future<partial_sort_copy_result<* range_iterator_t<Rng1>, range_iterator_t<Rng2>>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *partial_sort_copy_result<range_iterator_t<Rng1>*, range_iterator_t<Rng2>> otherwise. The algorithm returns {last, result_first + N}.

hpx::ranges::partition, hpx::ranges::stable_partition, hpx::ranges::partition_copy

Defined in header hpx/algorithm.hpp[644].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>> **partition**(*Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Reorders the elements in the range *rng* in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs at most 2 * N swaps, exactly N applications of the predicate and projection, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

[644] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition* algorithm returns *subrange_t<hpx::traits::range_iterator_t<Rng>>* The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to last.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>>> **partition**(*ExPolicy* &&policy, *Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Reorders the elements in the range *rng* in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs at most 2 * N swaps, exactly N applications of the predicate and projection, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition* algorithm returns a *hpx::future<subrange_t<hpx::traits::range_iterator_t<Rng>>>* if the execution policy is of type *parallel_task_policy* and returns *subrange_t<hpx::traits::range_iterator_t<Rng>>* The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to last.

template<typename **FwdIter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
subrange_t<*FwdIter*> **partition**(*FwdIter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: At most 2 * (last - first) swaps. Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition* algorithm returns returns *subrange_t<FwdIter>*. The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to last.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*FwdIter*>>::type **partition**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2 * (last - first) swaps. Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for

partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition* algorithm returns a *hpx::future<subrange_t<FwdIter>>* if the execution policy is of type *parallel_task_policy* and returns *subrange_t<FwdIter>* otherwise. The *partition* algorithm returns a subrange starting with an iterator to the first element of the second group and finishing with an iterator equal to last.

template<typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>> **stable_partition**(*Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) INVOKE(f, INVOKE (proj, *j)) != false, and for every iterator k in the range [i, last), INVOKE(f, INVOKE (proj, *k)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked without an execution policy object executes in sequential order in the calling thread.

Note: Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory Exactly *last - first* applications of the predicate and projection.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an birdirectional iterator
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range [first, i), f(*j) != false INVOKE(f, INVOKE(proj, *j)) != false, and for every iterator k in the range [i, last), f(*k) == false INVOKE(f, INVOKE (proj, *k)) == false. The relative order of the elements in both groups is preserved.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>>> **stable_partition**(*E*
&
ic
R
&
P
&
P
&
=
P

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) INVOKE(f, INVOKE (proj, *j)) != false, and for every iterator k in the range [i, last), INVOKE(f, INVOKE (proj, *k)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory. Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an birdirectional iterator
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range [first, i), f(*j) != false INVOKE(f, INVOKE(proj, *j)) != false, and for every iterator k in the range [i, last), f(*k) == false INVOKE(f, INVOKE (proj, *k)) == false. The relative order of the elements in both groups is preserved. If the execution policy is of type *parallel_task_policy* the algorithm returns a future<> referring to this iterator.

template<typename **BidirIter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
subrange_t<*BidirIter*> **stable_partition**(*BidirIter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) INVOKE(f, INVOKE (proj, *j)) != false, and for every iterator k in the range [i, last), INVOKE(f, INVOKE (proj, *k)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked without an execution policy object executes in sequential order in the calling thread.

Note: Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory Exactly *last* - *first* applications of the predicate and projection.

Template Parameters

- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for BidirIter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred** – Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range [first, i), f(*j) != false INVOKE(f, INVOKE(proj, *j)) != false, and for every iterator k in the range [i, last), f(*k) == false INVOKE(f, INVOKE (proj, *k)) == false. The relative order of the elements in both groups is preserved.

template<typename **ExPolicy**, typename **BidirIter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*BidirIter*>>::type **stable_partition**(*ExPolicy* &&policy, *BidirIter* first, *Sent* last, *Pred* &&pred, *Proj* &&proj = *Proj*())

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) INVOKE(f, INVOKE (proj, *j)) != false, and for every iterator k in the range [i, last), INVOKE(f, INVOKE (proj, *k)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) * log(last - first) swaps, but only linear number of swaps if there is enough extra memory Exactly *last - first* applications of the predicate and projection.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **BidirIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for BidirIter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred** – Unary predicate which returns true if the element should be ordered before other elements. Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *stable_partition* algorithm returns an iterator i such that for every iterator j in the range [first, i), f(*j) != false INVOKE(f, INVOKE(proj, *j)) != false, and for every iterator k in the range [i, last), f(*k) == false INVOKE(f, INVOKE (proj, *k)) == false. The relative order of the elements in both groups is preserved. If the execution policy is of type *parallel_task_policy* the algorithm returns a future<> referring to this iterator.

template<typename **Rng**, typename **OutIter2**, typename **OutIter3**, typename **Pred**, typename **Proj** = *hpx*::identity>
partition_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *OutIter2*, *OutIter3*> **partition_copy**(*Rng* &&rng, *OutIter2* dest_true, *OutIter3* dest_false, *Pred* &&pred, *Proj* &&proj = *Proj*())

Copies the elements in the range *rng*, to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs not more than N assignments, exactly N applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **OutIter2** – The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **OutIter3** – The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.

- **dest_true** – Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- **dest_false** – Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition_copy* algorithm returns a partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>>. The *partition_copy* algorithm returns the tuple of the source iterator *last*, the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, partition_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *FwdIter2*, *FwdI*

Copies the elements in the range *rng*, to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than N assignments, exactly N applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter3** – The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest_true** – Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- **dest_false** – Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition_copy* algorithm returns a *hpx::future<partition_copy_result* <hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>> if the execution policy is of type *parallel_task_policy* and returns partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3> otherwise. The *partition_copy* algorithm returns the tuple of the source iterator *last*, the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

template<typename **InIter**, typename **Sent**, typename **OutIter2**, typename **OutIter3**, typename **Pred**, typename **Proj** = *hpx*::identity>
partition_copy_result<*InIter*, *OutIter2*, *OutIter3*> **partition_copy**(*InIter* first, *Sent* last, *OutIter2* dest_true, *OutIter3* dest_false, *Pred* &&pred, *Proj* &&proj = *Proj*())

Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *f*.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **OutIter2** – The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **OutIter3** – The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest_true** – Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- **dest_false** – Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition_copy* algorithm returns a *partition_copy_result<FwdIter, OutIter2, OutIter3>*. The *partition_copy* algorithm returns the tuple of the source iterator *last*, the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **OutIter2**, typename **OutIter3**, typename **Pred**, typename **Proj** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, partition_copy_result<*FwdIter*, *OutIter2*, *OutIter3*>>::type **partition_co**

Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred* are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *f*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **OutIter2** – The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **OutIter3** – The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest_true** – Refers to the beginning of the destination range for the elements that satisfy the predicate *pred*
- **dest_false** – Refers to the beginning of the destination range for the elements that don't satisfy the predicate *pred*.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *partition_copy* algorithm returns a hpx::future<partition_copy_result<FwdIter, OutIter2, OutIter3>> if the execution policy is of type *parallel_task_policy* and returns *partition_copy_result<FwdIter, OutIter2, OutIter3>* otherwise. The *partition_copy* algorithm returns the tuple of the source iterator *last*, the destination iterator to the end of the *dest_true* range, and the destination iterator to the end of the *dest_false* range.

hpx::ranges::reduce

Defined in header hpx/algorithm.hpp[645].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **F**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **reduce**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *T* init, *F* &&f)

Returns GENERALIZED_SUM(f, init, *first, …, *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

[645] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicate *f*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.

- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **reduce**(*ExPolicy* &&policy, *Rng* &&rng, *T* init, *F* &&f)

Returns GENERALIZED_SUM(f, init, *first, ..., *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicate *f*.

Note: GENERALIZED_SUM(op, a1, ..., aN) is defined as follows:
- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, ..., bK), GENERALIZED_SUM(op, bM, ..., bN)), where:
  - b1, ..., bN may be any permutation of a1, ..., aN and
  - 1 < K+1 = M <= N.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>

hpx::parallel::util::detail::algorithm_result_t<ExPolicy, T> **reduce**(ExPolicy &&policy, FwdIter first, Sent last, T init)

Returns GENERALIZED_SUM(+, init, *first, …, *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **Rng**, typename **T** = typename std::iterator_traits<hpx::traits::range_iterator_t<Rng>>::value_type>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, T> **reduce**(ExPolicy &&policy, Rng &&rng, T init)

Returns GENERALIZED_SUM(+, init, *first, …, *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<*FwdIter*>::value_type>::type **reduce**(*E*
&
i
F
fi
S
la

Returns GENERALIZED_SUM(+, T(), *first, …, *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the value_type of the used *FwdIterB*.

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns T otherwise (where T is the value_type of *FwdIterB*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **Rng**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::iterator_traits<typename *hpx*::*traits*::range_traits<*Rng*

Returns GENERALIZED_SUM(+, T(), *first, …, *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the value_type of the used *FwdIterB*.

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns T otherwise (where T is the value_type of *FwdIterB*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **FwdIter**, typename **Sent**, typename **F**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
T **reduce**(*FwdIter* first, *Sent* last, *T* init, *F* &&f)

Returns GENERALIZED_SUM(f, init, *first, …, *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicate *f*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent** – The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns *T*. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

template<typename **Rng**, typename **F**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
T **reduce**(*Rng* &&rng, *T* init, *F* &&f)

Returns GENERALIZED_SUM(f, init, *first, …, *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicate *f*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.

- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns *T*. The *reduce* algorithm returns the result of the generalized sum over the elements given by the input range [first, last).

template<typename **FwdIter**, typename **Sent**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type>
T **reduce**(*FwdIter* first, *Sent* last, *T* init)

Returns GENERALIZED_SUM(+, init, *first, . . . , *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, . . . , aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, . . . , bK), GENERALIZED_SUM(+, bM, . . . , bN)), where:
  - b1, . . . , bN may be any permutation of a1, . . . , aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns *T*. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **Rng**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
T **reduce**(*Rng* &&rng, *T* init)

Returns GENERALIZED_SUM(+, init, *first, . . . , *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *reduce* algorithm returns *T*. The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **FwdIter**, typename **Sent**>
std::iterator_traits<*FwdIter*>::value_type **reduce**(*FwdIter* first, *Sent* last)

Returns GENERALIZED_SUM(+, T(), *first, …, *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the value_type of the used *FwdIterB*.

Note: GENERALIZED_SUM(+, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, …, bK), GENERALIZED_SUM(+, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **FwdIter** – The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns *T* (where T is the value_type of *FwdIterB*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

template<typename **Rng**>

std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_type>::value_type **reduce**(Rng &&rng)

Returns GENERALIZED_SUM(+, T(), *first, ..., *(first + (last - first) - 1)).

The difference between *reduce* and *accumulate* is that the behavior of reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the operator+().

Note: The type of the initial value (and the result type) *T* is determined from the value_type of the used *FwdIterB*.

Note: GENERALIZED_SUM(+, a1, ..., aN) is defined as follows:
- a1 when N is 1
- op(GENERALIZED_SUM(+, b1, ..., bK), GENERALIZED_SUM(+, bM, ..., bN)), where:
  - b1, ..., bN may be any permutation of a1, ..., aN and
  - 1 < K+1 = M <= N.

Template Parameters **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters **rng** – Refers to the sequence of elements the algorithm will be applied to.

Returns The *reduce* algorithm returns *T* (where T is the value_type of *FwdIterB*). The *reduce* algorithm returns the result of the generalized sum (applying operator+()) over the elements given by the input range [first, last).

hpx::ranges::remove, hpx::ranges::remove_if

Defined in header hpx/algorithm.hpp[646].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **Iter**, typename **Sent**, typename **Pred**, typename **Proj** = hpx::identity>
subrange_t<Iter, Sent> **remove_if**(Iter first, Sent sent, Pred &&pred, Proj &&proj = Proj())

Removes all elements for which predicate *pred* returns true from the range [first, last) and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm execute in sequential order in the calling thread.

[646] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred* and the projection *proj*.

Template Parameters

- **Iter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*..
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove_if* algorithm returns a *subrange_t<FwdIter, Sent>*. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

template<typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>> **remove_if**(*Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Removes all elements that are equal to *value* from the range *rng* and and returns a subrange [ret, util::end(rng)), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *util::end(rng)*

- *util::begin(rng)* assignments, exactly *util::end(rng)* - *util::begin(rng)* applications of the operator==() and the projection *proj*.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove_if* algorithm returns a *subrange_t<hpx::traits::range_iterator_t<Rng>>*. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*FwdIter*, *Sent*>>::type **remove_if**(*ExPolicy* &&policy, *FwdIter* first, *Sent* sent, *Pred* &&pred, *Proj* &&proj = *Proj*())

Removes all elements for which predicate *pred* returns true from the range [first, last) and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the predicate *pred* and the projection *proj*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.

- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove_if* algorithm returns a *hpx::future<subrange_t<FwdIter, Sent>>*. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>>> **remove_if**(*ExPolicy* &&policy, *Rng* &&rng, *Pred* &&pred, *Proj* &&proj = *Proj*())

Removes all elements that are equal to *value* from the range *rng* and and returns a subrange [ret, util::end(rng)), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *util::end(rng)*

- *util::begin(rng)* assignments, exactly *util::end(rng)* - *util::begin(rng)* applications of the operator==() and the projection *proj*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it

executes the assignments.

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *remove_if* requires *Pred* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove_if* algorithm returns a *hpx::future<subrange_t<* hpx::traits::range_iterator_t<Rng>>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove_if* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

template<typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type>
subrange_t<*Iter*, *Sent*> **remove**(*Iter* first, *Sent* last, *T* const &value, *Proj* &&proj = *Proj*())

Removes all elements that are equal to *value* from the range [first, last) and and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the operator==() and the projection *proj*.

Template Parameters

- **Iter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **T** – The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – Specifies the value of elements to remove.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove* algorithm returns a *subrange_t<FwdIter, Sent>*. The *remove* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

template<typename **Rng**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>
subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>> **remove**(*Rng* &&rng, *T* const &value, *Proj* &&proj = *Proj*())

Removes all elements that are equal to *value* from the range *rng* and and returns a subrange [ret, util::end(rng)), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *util::end(rng)*

- *util::begin(rng)* assignments, exactly *util::end(rng)* - *util::begin(rng)* applications of the operator==() and the projection *proj*.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **T** – The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – Specifies the value of elements to remove.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove* algorithm returns a *subrange_t<hpx::traits::range_iterator_t<Rng>>*. The *remove* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*FwdIter*, *Proj*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*FwdIter*, *Sent*>>::type **remove**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *T* const &value, *Proj* &&proj = *Proj*())

Removes all elements that are equal to *value* from the range [first, last) and and returns a subrange [ret, last), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* applications of the operator==() and the projection *proj*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **T** – The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **value** – Specifies the value of elements to remove.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove* algorithm returns a *hpx::future<subrange_t<FwdIter, Sent>>*. The *remove* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange of the values all in valid but unspecified state.

template<typename **ExPolicy**, typename **Rng**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>>> **remove**(*ExPolicy* &&policy, *Rng* &&rng, *T* const &value, *Proj* &&proj = *Proj*())

Removes all elements that are equal to *value* from the range *rng* and and returns a subrange [ret, util::end(rng)), where ret is a past-the-end iterator for the new end of the range.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *remove* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *util::end(rng)*

- *util::begin(rng)* assignments, exactly *util::end(rng)* - *util::begin(rng)* applications of the operator==() and the projection *proj*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **T** – The type of the value to remove (deduced). This value type must meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **value** – Specifies the value of elements to remove.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *remove* algorithm returns a *hpx::future*< subrange_t<hpx::traits::range_iterator_t<Rng>>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove* algorithm returns the iterator to the new end of the range.

hpx::ranges::remove_copy, hpx::ranges::remove_copy_if

Defined in header hpx/algorithm.hpp[647].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

hpx::ranges::replace, hpx::ranges::replace_if, hpx::ranges::replace_copy, hpx::ranges::replace_copy_if

Defined in header hpx/algorithm.hpp[648].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[647] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp
[648] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **Iter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type>
Iter **replace_if**(*Iter* first, *Sent* sent, *Pred* &&pred, *T* const &new_value, *Proj* &&proj = *Proj*())

Replaces all elements satisfying specific criteria (for which predicate *f* returns true) with *new_value* in the range [first, sent).

Effects: Substitutes elements referred by the iterator it in the range [first, sent) with new_value, when the following corresponding conditions hold: INVOKE(f, INVOKE(proj, *it)) != false

The assignments in the parallel *replace_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *sent* - *first* applications of the predicate.

Template Parameters

- **Iter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_if* algorithm returns *Iter*. It returns *last*.

template<typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>
hpx::*traits*::range_iterator_t<*Rng*> **replace_if**(*Rng* &&rng, *Pred* &&pred, *T* const &new_value, *Proj* &&proj = *Proj*())

Replaces all elements satisfying specific criteria (for which predicate *pred* returns true) with *new_value* in the range rng.

Effects: Substitutes elements referred by the iterator it in the range rng with new_value, when the following corresponding conditions hold: INVOKE(f, INVOKE(proj, *it)) != false

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by rng.This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_if* algorithm returns an *hpx::traits::range_iterator<Rng>::type*. It returns *last*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Pred**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **replace_if**(*ExPolicy* &&policy, *Iter* first, *Sent* sent, *Pred* &&pred, *T* const &new_value, *Proj* &&proj = *Proj*())

Replaces all elements satisfying specific criteria (for which predicate *pred* returns true) with *new_value* in the range rng.

Effects: Substitutes elements referred by the iterator it in the range rng with new_value, when the following corresponding conditions hold: INVOKE(f, INVOKE(proj, *it)) != false

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_if* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy*. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *hpx*::identity, typename **T** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **replace_if**(*ExPolicy* &&policy, *Rng* &&rng, *Pred* &&pred, *T* const &new_value, *Proj* &&proj = *Proj*())

Replaces all elements satisfying specific criteria (for which predicate *pred* returns true) with *new_value* in the range rng.

Effects: Substitutes elements referred by the iterator it in the range rng with new_value, when the following corresponding conditions hold: INVOKE(f, INVOKE(proj, *it)) != false

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *F* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by rng.This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

  ```
  bool pred(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_if* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy*. It returns *last*.

template<typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type, typename **T2** = *T1*>
Iter **replace**(*Iter* first, *Sent* sent, *T1* const &old_value, *T2* const &new_value, *Proj* &&proj = *Proj*())

Replaces all elements satisfying specific criteria with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first,last) with new_value, when the following corresponding conditions hold: INVOKE(proj, *i) == old_value

The assignments in the parallel *replace* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **`Iter`** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **`Sent`** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **`T1`** – The type of the old value to replace (deduced).
- **`T2`** – The type of the new values to replace (deduced).
- **`Proj`** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **`sent`** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **`old_value`** – Refers to the old value of the elements to replace.
- **`new_value`** – Refers to the new value to use as the replacement.
- **`proj`** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace* algorithm returns an *Iter*.

template<typename **Rng**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type, typename **T2** = *T1*>
hpx::*traits*::range_iterator_t<*Rng*> **replace**(*Rng* &&rng, *T1* const &old_value, *T2* const &new_value, *Proj* &&proj = *Proj*())

Replaces all elements satisfying specific criteria with *new_value* in the range *rng*.

Effects: Substitutes elements referred by the iterator it in the range rng with new_value, when the following corresponding conditions hold: INVOKE(proj, *i) == old_value

The assignments in the parallel *replace* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* assignments.

Template Parameters

- **`Rng`** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **`T1`** – The type of the old value to replace (deduced).
- **`T2`** – The type of the new values to replace (deduced).
- **`Proj`** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **`rng`** – Refers to the sequence of elements the algorithm will be applied to.
- **`old_value`** – Refers to the old value of the elements to replace.
- **`new_value`** – Refers to the new value to use as the replacement.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace* algorithm returns an *hpx::traits::range_iterator<Rng>::type*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*Iter*, *Proj*>::value_type, typename **T2** = *T1*>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **replace**(*ExPolicy* &&policy, *Iter* first, *Sent* sent, *T1* const &old_value, *T2* const &new_value, *Proj* &&proj = *Proj*())

Replaces all elements satisfying specific criteria with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first,last) with new_value, when the following corresponding conditions hold: INVOKE(proj, *i) == old_value

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **T1** – The type of the old value to replace (deduced).
- **T2** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise.

template<typename **ExPolicy**, typename **Rng**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type, typename **T2** = *T1*>

parallel::util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng>> **replace**(ExPolicy &&policy, Rng &&rng, T1 const &old_value, T2 const &new_value, Proj &&proj = Proj())

Replaces all elements satisfying specific criteria with *new_value* in the range *rng*.

Effects: Substitutes elements referred by the iterator it in the range rng with new_value, when the following corresponding conditions hold: INVOKE(proj, *i) == old_value

The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **T1** – The type of the old value to replace (deduced).
- **T2** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked. The assignments in the parallel *replace* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Returns The *replace* algorithm returns an *hpx::future<hpx::traits::range_iterator<Rng>::type>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::traits::range_iterator<Rng>::type* otherwise.

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **Pred**, typename **T** = typename std::iterator_traits<OutIter>::value_type, typename **Proj** = hpx::identity>
replace_copy_if_result<InIter, OutIter> **replace_copy_if**(InIter first, Sent sent, OutIter dest, Pred &&pred, T const &new_value, Proj &&proj = Proj())

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (sent - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(f, INVOKE(proj, *(first + (i - result)))) != false

The assignments in the parallel *replace_copy_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *sent* - *first* applications of the predicate.

Template Parameters

- **InIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for InIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy_if* algorithm returns a *in_out_result<InIter, OutIter>*. The *replace_copy_if* algorithm returns the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **OutIter**, typename **Pred**, typename **T** = typename *std*::iterator_traits<*OutIter*>::value_type, typename **Proj** = *hpx*::identity>
replace_copy_if_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *OutIter*> **replace_copy_if**(*Rng* &&rng, *OutIter* dest, *Pred* &&pred, *T* const &new_value, *Proj* &&proj = *Proj*())

Copies the all elements from the range rng to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (util::end(rng) - util::begin(rng))) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(f, INVOKE(proj, *(first + (i - result)))) != false

The assignments in the parallel *replace_copy_if* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy_if* algorithm returns an *in_out_result<hpx::traits::range_iterator_t<Rng>*, OutIter>. The *replace_copy_if* algorithm returns the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **Pred**, typename **T** = typename *std*::iterator_traits<*FwdIter2*>::value_type, typename **Proj** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, replace_copy_if_result<*FwdIter1*, *FwdIter2*>>::type **`replace_copy_if`**(

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (sent - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(f, INVOKE(proj, *(first + (i - result)))) != false

The assignments in the parallel *replace_copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *sent* - *first* applications of the predicate.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **`FwdIter1`** – The type of the source iterator used (deduced). The iterator type must meet the requirements of a forward iterator.
- **`Sent`** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for InIter.
- **`FwdIter2`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **`Pred`** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **`T`** – The type of the new values to replace (deduced).
- **`Proj`** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **`policy`** – The execution policy to use for the scheduling of the iterations.
- **`first`** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy_if* algorithm returns an *hpx::future<FwdIter1, FwdIter2>*. The *replace_copy_if* algorithm returns the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter**, typename **Pred**, typename **T** = typename *std*::iterator_traits<*FwdIter*>::value_type, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, replace_copy_if_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *FwdIter*>>::typ

Copies the all elements from the range rng to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (util::end(rng) - util::begin(rng))) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(f, INVOKE(proj, *(first + (i - result)))) != false

The assignments in the parallel *replace_copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_copy_if* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. (deduced).
- **T** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy_if* algorithm returns an *hpx::future<in_out_result<hpx::traits::range_iterator_t<Rng>* OutIter>>. The *replace_copy_if* algorithm returns the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*InIter*, *Proj*>::value_type, typename **T2** = *T1*>
replace_copy_result<*InIter*, *OutIter*> **replace_copy**(*InIter* first, *Sent* sent, *OutIter* dest, *T1* const &old_value, *T2* const &new_value, *Proj* &&proj = *Proj*())

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (sent - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(proj, *(first + (i - result))) == old_value

The assignments in the parallel *replace_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *sent - first* applications of the predicate.

Template Parameters

- **InIter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T1** – The type of the old value to replace (deduced).
- **T2** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy* algorithm returns an *in_out_result<InIter, OutIter>*. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **OutIter**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type, typename **T2** = *T1*>
replace_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *OutIter*> **replace_copy**(*Rng* &&rng, *OutIter* dest, *T1* const &old_value, *T2* const &new_value, *Proj* &&proj = *Proj*())

Copies the all elements from the range rbg to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (util::end(rng) - util::begin(rng))) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(proj, *(first + (i - result))) == old_value

The assignments in the parallel *replace_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T1** – The type of the old value to replace (deduced).
- **T2** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.

- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy* algorithm returns an *in_out_result<hpx::traits::range_iterator_t<Rng>*, OutIter>. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*FwdIter1*, *Proj*>::value_type, typename **T2** = *T1*>
parallel::*util*::detail::algorithm_result<*ExPolicy*, replace_copy_result<*FwdIter1*, *FwdIter2*>>::type **replace_copy**(*ExPolic* &&policy, *FwdIte* first, *Sent* sent, *FwdIte* dest, *T1* const &old_v *T2* const &new_ *Proj* &&pro = *Proj*())

Copies the all elements from the range [first, sent) to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (sent - first)) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(proj, *(first + (i - result))) == old_value

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *sent* - *first* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an forward iterator.

- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T1** – The type of the old value to replace (deduced).
- **T2** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy* algorithm returns a *hpx::future<in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<FwdIter1, FwdIter2>* otherwise. The *copy* algorithm returns the pair of the forward iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter**, typename **Proj** = *hpx*::identity, typename **T1** = typename *hpx*::*parallel*::traits::projected<*hpx*::*traits*::range_iterator_t<*Rng*>, *Proj*>::value_type, typename **T2** = *T1*>
parallel::*util*::detail::algorithm_result<*ExPolicy*, replace_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *FwdIter*>>::type

Copies the all elements from the range rbg to another range beginning at *dest* replacing all elements satisfying a specific criteria with *new_value*.

Effects: Assigns to every iterator it in the range [result, result + (util::end(rng) - util::begin(rng))) either new_value or *(first + (it - result)) depending on whether the following corresponding condition holds: INVOKE(proj, *(first + (i - result))) == old_value

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *replace_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *util::end(rng) - util::begin(rng)* applications of the predicate.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T1** – The type of the old value to replace (deduced).
- **T2** – The type of the new values to replace (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **old_value** – Refers to the old value of the elements to replace.
- **new_value** – Refers to the new value to use as the replacement.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *replace_copy* algorithm returns a *hpx::future<in_out_result<* hpx::traits::range_iterator_t<Rng>, FwdIter>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<* hpx::traits::range_iterator_t<Rng>, FwdIter>> The *copy* algorithm returns the pair of the input iterator *last* and the forward iterator to the element in the destination range, one past the last element copied.

hpx::ranges::reverse, hpx::ranges::reverse_copy

Defined in header hpx/algorithm.hpp[649].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[649] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **Iter**, typename **Sent**>
Iter **reverse**(*Iter* first, *Sent* sent)

Reverses the order of the elements in the range [first, last). Behaves as if applying std::iter_swap to every pair of iterators first+i, (last-i) - 1 for each non-negative i < (last-first)/2.

The assignments in the parallel *reverse* algorithm execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **Iter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reverse* algorithm returns a *Iter*. It returns *last*.

template<typename **Rng**>
hpx::*traits*::range_iterator_t<*Rng*> **reverse**(*Rng* &&rng)

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Reverses the order of the elements in the range [first, last). Behaves as if applying std::iter_swap to every pair of iterators first+i, (last-i) - 1 for each non-negative i < (last-first)/2.

The assignments in the parallel *reverse* algorithm execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters Rng – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a bidirectional iterator.

Parameters rng – Refers to the sequence of elements the algorithm will be applied to.

Returns The *reverse* algorithm returns a *hpx::traits::range_iterator<Rng>::type*. It returns *last*.

template<typename **ExPolicy**, typename **Iter**, typename **Sent**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *Iter*> **reverse**(*ExPolicy* &&policy, *Iter* first, *Sent* sent)

Reverses the order of the elements in the range [first, last). Behaves as if applying std::iter_swap to every pair of iterators first+i, (last-i) - 1 for each non-negative i < (last-first)/2.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **sent** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *reverse* algorithm returns a *hpx::future<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Rng**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **reverse**(*ExPolicy* &&policy, *Rng* &&rng)

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Reverses the order of the elements in the range [first, last). Behaves as if applying std::iter_swap to every pair of iterators first+i, (last-i) - 1 for each non-negative i < (last-first)/2.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *reverse* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a bidirectional iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.

Returns The *reverse* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::future*< hpx::traits::range_iterator_t<Rng>> otherwise. It returns *last*.

template<typename **Iter**, typename **Sent**, typename **OutIter**>

reverse_copy_result<*Iter*, *OutIter*> **reverse_copy**(*Iter* first, *Sent* last, *OutIter* result)

Copies the elements from the range [first, last) to another range beginning at result in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(result + (last - first) - 1 - i) = *(first + i) once for each non-negative i < (last - first) If the source and destination ranges (that is, [first, last) and [result, result+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **Iter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **result** – Refers to the begin of the destination range.

Returns The *reverse_copy* algorithm returns a *reverse_copy_result<Iter, OutIter>*. The *reverse_copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **OutIter**>
reverse_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *OutIter*> **reverse_copy**(*Rng* &&rng, *OutIter* result)

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Copies the elements from the range [first, last) to another range beginning at result in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(result + (last - first) - 1 - i) = *(first + i) once for each non-negative i < (last - first) If the source and destination ranges (that is, [first, last) and [result, result+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a bidirectional iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **result** – Refers to the begin of the destination range.

Returns The *reverse_copy* algorithm returns a *ranges::reverse_copy_result*< hpx::traits::range_iterator_t<Rng>, OutIter>. The *reverse_copy* algorithm returns an object equal to {last, result + N} where N = last - first

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **FwdIter**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, reverse_copy_result<*Iter*, *FwdIter*>>::type **reverse_copy**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *FwdIter* result)

Copies the elements from the range [first, last) to another range beginning at result in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(result + (last - first) - 1 - i) = *(first + i) once for each non-negative i < (last - first) If the source and destination ranges (that is, [first, last) and [result, result+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterator used (deduced). The iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for Iter.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **result** – Refers to the begin of the destination range.

Returns The *reverse_copy* algorithm returns a *hpx::future<reverse_copy_result<Iter, FwdIter> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *reverse_copy_result<Iter, FwdIter>* otherwise. The *reverse_copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, reverse_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *OutIter*>>::type **r**

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Copies the elements from the range [first, last) to another range beginning at result in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(result + (last - first) - 1 - i) = *(first + i) once for each non-negative i < (last - first) If the source and destination ranges (that is, [first, last) and [result, result+(last-first)) respectively) overlap, the behavior is undefined.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *reverse_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a bidirectional iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **result** – Refers to the begin of the destination range.

Returns The *reverse_copy* algorithm returns a *hpx::future<ranges::reverse_copy_result<* hpx::traits::range_iterator_t<Rng>, OutIter>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::reverse_copy_result<* hpx::traits::range_iterator_t<Rng>, OutIter> otherwise. The *reverse_copy* algorithm returns an object equal to {last, result + N} where N = last - first

hpx::ranges::rotate, hpx::ranges::rotate_copy

Defined in header hpx/algorithm.hpp[650].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**>
subrange_t<*FwdIter*, *Sent*> **rotate**(*FwdIter* first, *FwdIter* middle, *Sent* last)

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element middle becomes the first element of the new range and middle - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Template Parameters
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *rotate* algorithm returns a *subrange_t<FwdIter, Sent>*. The *rotate* algorithm returns the iterator equal to pair(first + (last - middle), last).

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*FwdIter*, *Sent*>>::type **rotate**(*ExPolicy* &&policy, *FwdIter* first, *FwdIter* middle, *Sent* last)

Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range

650 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

[first, last) in such a way that the element middle becomes the first element of the new range and middle - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.

Returns The *rotate* algorithm returns a *hpx::future<subrange_t<FwdIter, Sent>>* if the execution policy is of type *parallel_task_policy* and returns a *subrange_t<FwdIter, Sent>* otherwise. The *rotate* algorithm returns the iterator equal to pair(first + (last - middle), last).

template<typename **Rng**>
subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>, *hpx*::*traits*::range_iterator_t<*Rng*>> **rotate**(*Rng* &&rng, *hpx*::*traits*::range_iterator_t<*R* middle)

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element middle becomes the first element of the new range and middle - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Template Parameters **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.

Returns The *rotate* algorithm returns a *subrange_t<hpx::traits::range_iterator_t<Rng>*, hpx::traits::range_iterator_t<Rng>>. The *rotate* algorithm returns the iterator equal to pair(first + (last - middle), last).

template<typename **ExPolicy**, typename **Rng**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>, *hpx*::*traits*::range_iterator_

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Performs a left rotation on a range of elements. Specifically, *rotate* swaps the elements in the range [first, last) in such a way that the element middle becomes the first element of the new range and middle - 1 becomes the last element.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *rotate* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable* and *MoveConstructible*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.

Returns The *rotate* algorithm returns a *hpx::future* <subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *subrange_t<hpx::traits::range_iterator_t<Rng>*, hpx::traits::range_iterator_t<Rng>>. otherwise. The *rotate* algorithm returns the iterator equal to pair(first + (last - middle), last).

template<typename **FwdIter**, typename **Sent**, typename **OutIter**>
rotate_copy_result<*FwdIter*, *OutIter*> **rotate_copy**(*FwdIter* first, *FwdIter* middle, *Sent* last, *OutIter* dest_first)

Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *middle* becomes the first element of the new range and *middle* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest_first** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first,last).

Returns The *rotate_copy* algorithm returns a *rotate_copy_result<FwdIter, OutIter>*. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, rotate_copy_result<*FwdIter1*, *FwdIter2*>>::type **rotate_copy**(*ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* middle, *Sent* last, *FwdIter2* dest_first)

Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *middle* becomes the first element of the new range and *middle* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the end iterators used (deduced). This sentinel type must be a sentinel for FwdIter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest_first** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first,last).

Returns The *rotate_copy* algorithm returns areturns hpx::future< rotate_copy_result<FwdIter1, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *rotate_copy_result<FwdIter1, FwdIter2>* otherwise. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

template<typename **Rng**, typename **OutIter**>
rotate_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *OutIter*> **rotate_copy**(*Rng* &&rng, *hpx*::*traits*::range_iterator_t<*Rng*> middle, *OutIter* dest_first)

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *middle* becomes the first element of the new range and *middle* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.
- **dest_first** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first,last).

Returns The *rotate* algorithm returns a *rotate_copy_result<hpx::traits::range_iterator_t<Rng>*, OutIter>. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, rotate_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *OutIter*>> **rotate_**

Uses *rng* as the source range, as if using *util::begin(rng)* as *first* and *ranges::end(rng)* as *last*. Copies the elements from the range [first, last), to another range beginning at *dest_first* in such a way, that the element *new_first* becomes the first element of the new range and *new_first* - 1 becomes the last element.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *rotate_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **middle** – Refers to the element that should appear at the beginning of the rotated range.
- **dest_first** – Output iterator to the initial position of the range where the reversed range is stored. The pointed type shall support being assigned the value of an element in the range [first,last).

Returns The *rotate_copy* algorithm returns a *hpx::future<otate_copy_result<* hpx::traits::range_iterator_t<Rng>, OutIter>> if the execution policy is of type *par-*

allel_task_policy and returns *rotate_copy_result*< hpx::traits::range_iterator_t<Rng>, OutIter> otherwise. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

hpx::ranges::search, hpx::ranges::search_n

Defined in header hpx/algorithm.hpp[651].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**, typename **FwdIter2**, typename **Sent2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
FwdIter **search**(*FwdIter* first, *Sent* last, *FwdIter2* s_first, *Sent2* s_last, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel used for the first range (deduced). This iterator type must meet the requirements of an sentinel.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter2*.

Parameters
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.

[651] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter2* as a projection operation before the actual predicate *is* invoked.

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **FwdIter2**, typename **Sent2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **search**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *FwdIter2* s_first, *Sent2* s_last, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it

executes the assignments.

- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel used for the first range (deduced). This iterator type must meet the requirements of an sentinel.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter2* as a projection operation before the actual predicate *is* invoked.

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng1*> **search**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **Rng1** – The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng2*.

Parameters

- **rng1** – Refers to the sequence of elements the algorithm will be examining.
- **rng2** – Refers to the sequence of elements the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng1>> **search**(ExPolicy &&policy, Rng1 &&rng1, Rng2 &&rng2, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the sequence of elements the algorithm will be examining.
- **rng2** – Refers to the sequence of elements the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated

as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

template<typename **FwdIter**, typename **FwdIter2**, typename **Sent2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
FwdIter **search_n**(*FwdIter* first, *std*::size_t count, *FwdIter2* s_first, Sent s_last, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search_n* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = count.

Template Parameters

- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter2*.

Parameters

- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **count** – Refers to the range of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.

- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter2* as a projection operation before the actual predicate *is* invoked.

Returns The *search_n* algorithm returns *FwdIter*. The *search_n* algorithm returns an iterator to the beginning of the last subsequence [s_first, s_last) in range [first, first+count). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, first+count), *first* is returned. Additionally, if the size of the subsequence is empty or no subsequence is found, *first* is also returned.

template<typename **ExPolicy**, typename **FwdIter**, typename **FwdIter2**, typename **Sent2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **search_n**(*ExPolicy* &&policy, *FwdIter* first, *std*::size_t count, *FwdIter2* s_first, *Sent2* s_last, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *search_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = count.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2** – The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent2** – The type of the source sentinel used for the second range (deduced). This iterator type must meet the requirements of an sentinel.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of type dereferenced *FwdIter2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **count** – Refers to the range of elements of the first range the algorithm will be applied to.
- **s_first** – Refers to the beginning of the sequence of elements the algorithm will be searching for.
- **s_last** – Refers to the end of the sequence of elements of the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced *FwdIter2* as a projection operation before the actual predicate *is* invoked.

Returns The *search_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search_n* algorithm returns an iterator to the beginning of the last subsequence [s_first, s_last) in range [first, first+count). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, first+count), *first* is returned. Additionally if the size of the subsequence is empty or no subsequence is found, *first* is also returned.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng1*> **search_n**(*Rng1* &&rng1, *std*::size_t count, *Rng2* &&rng2, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm execute in sequential order in the calling thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **Rng1** – The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng2*.

Parameters

- **rng1** – Refers to the sequence of elements the algorithm will be examining.
- **count** – The number of elements to apply the algorithm on.
- **rng2** – Refers to the sequence of elements the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *hpx*::*ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng1>> **search_n**(ExPolicy &&policy, Rng1 &&rng1, std::size_t count, Rng2 &&rng2, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj1** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng1*.
- **Proj2** – The type of an optional projection function. This defaults to *hpx::identity* and is applied to the elements of *Rng2*.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the sequence of elements the algorithm will be examining.

- **count** – The number of elements to apply the algorithm on.
- **rng2** – Refers to the sequence of elements the algorithm will be searching for.
- **op** – Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

  ```
  bool pred(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

Returns The *search* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *task_execution_policy* and returns *FwdIter* otherwise. The *search* algorithm returns an iterator to the beginning of the first subsequence [s_first, s_last) in range [first, last). If the length of the subsequence [s_first, s_last) is greater than the length of the range [first, last), *last* is returned. Additionally if the size of the subsequence is empty *first* is returned. If no subsequence is found, *last* is returned.

hpx::ranges::set_difference

Defined in header hpx/algorithm.hpp[652].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[652] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_difference_result<*Iter1*, *Iter3*>>::type **set_difference**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_difference* algorithm returns a *hpx::future<ranges::set_difference_result<Iter1, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_difference_result<Iter1, Iter3>* otherwise. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_difference_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *Iter3*>> **se**

Constructs a sorted range beginning at dest consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_difference* algorithm returns a *hpx::future<ranges::set_difference_result<Iter1, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_difference_result<Iter1, Iter3>* otherwise. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
set_difference_result<*Iter1*, *Iter3*> **set_difference**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_difference* algorithm returns *ranges::set_difference_result<Iter1, Iter3>*. The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
set_difference_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *Iter3*> **set_difference**(*Rng1* &&rng1, *Rng2* &&rng2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in the range [first1, last1) and not present in the range [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

Equivalent elements are treated individually, that is, if some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::max(m-n, 0) times. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_difference* algorithm returns *ranges::set_difference_result<Iter1, Iter3>*. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::set_intersection

Defined in header hpx/algorithm.hpp[653].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_intersection_result<*Iter1*, *Iter2*, *Iter3*>>::type **set_intersectio**

Constructs a sorted range beginning at dest consisting of all elements present in both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), the first std::min(m, n) elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

[653] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.

- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_intersection* algorithm returns a *hpx::future<ranges::set_intersection_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_intersection_result<Iter1, Iter2, Iter3>* otherwise. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_intersection_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*trai*

Constructs a sorted range beginning at dest consisting of all elements present in both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), the first std::min(m, n) elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_intersection* algorithm returns a *hpx::future<ranges::set_intersection_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_intersection_result<Iter1, Iter2, Iter3>* otherwise. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
set_intersection_result<*Iter1*, *Iter2*, *Iter3*> **set_intersection**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), the first std::min(m, n) elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **`Iter1`** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **`Sent1`** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **`Iter2`** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **`Sent2`** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **`Iter3`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **`Pred`** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **`Proj1`** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **`Proj2`** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **`first1`** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **`last1`** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **`first2`** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **`last2`** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **`dest`** – Refers to the beginning of the destination range.
- **`op`** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **`proj1`** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **`proj2`** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_intersection* algorithm returns *ranges::set_intersection_result<Iter1, Iter2, Iter3>*. The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
set_intersection_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range_iterator_t<*Rng2*>, *Iter3*> **set_intersectio**

Constructs a sorted range beginning at dest consisting of all elements present in both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), the first std::min(m, n) elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_intersection* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_intersection* algorithm returns *ranges::set_intersection_result<Iter1, Iter2, Iter3>*. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_intersection* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::set_symmetric_difference

Defined in header hpx/algorithm.hpp[654].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

[654] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_symmetric_difference_result<*Iter1*, *Iter2*, *Iter3*>>::type **set_sym**

Constructs a sorted range beginning at dest consisting of all elements present in either of the sorted ranges [first1, last1) and [first2, last2), but not in both of them are copied to the range beginning at *dest*. The resulting range is also sorted. This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::abs(m-n) times. If m>n, then the last m-n of those elements are copied from [first1,last1), otherwise the last n-m elements are copied from [first2,last2). The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.

- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_symmetric_difference* algorithm returns a *hpx::future<ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>* otherwise. The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_symmetric_difference_result<*hpx*::*traits*::range_iterator_t<*Rng1*>

Constructs a sorted range beginning at dest consisting of all elements present in either of the sorted ranges [first1, last1) and [first2, last2), but not in both of them are copied to the range beginning at *dest*. The resulting range is also sorted. This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::abs(m-n) times. If m>n, then the last m-n of those elements are copied from [first1,last1), otherwise the last n-m elements are copied from [first2,last2). The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **`ExPolicy`** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **`Rng1`** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **`Rng2`** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_symmetric_difference* algorithm returns a *hpx::future<ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>* otherwise. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
set_symmetric_difference_result<*Iter1*, *Iter2*, *Iter3*> **set_symmetric_difference**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in either of the sorted ranges [first1, last1) and [first2, last2), but not in both of them are copied to the range beginning at *dest*. The resulting range is also sorted. This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::abs(m-n) times. If m>n, then the last m-n of those elements are copied from [first1,last1), otherwise the last n-m elements are copied from [first2,last2). The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **`Iter1`** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **`Sent1`** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **`Iter2`** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **`Sent2`** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **`Iter3`** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **`Pred`** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **`Proj1`** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **`Proj2`** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **`first1`** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **`last1`** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **`first2`** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **`last2`** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **`dest`** – Refers to the beginning of the destination range.
- **`op`** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **`proj1`** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **`proj2`** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_symmetric_difference* algorithm returns *ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>*. The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
set_symmetric_difference_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range_iterator_t<*Rng2*>, *Iter3*> **set_sym**

Constructs a sorted range beginning at dest consisting of all elements present in either of the sorted ranges [first1, last1) and [first2, last2), but not in both of them are copied to the range beginning at *dest*. The resulting range is also sorted. This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), it will be copied to *dest* exactly std::abs(m-n) times. If m>n, then the last m-n of those elements are copied from [first1,last1), otherwise the last n-m elements are copied from [first2,last2). The resulting range cannot overlap with either of the input ranges.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_symmetric_difference* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.

- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_symmetric_difference* algorithm returns *ranges::set_symmetric_difference_result<Iter1, Iter2, Iter3>*. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_symmetric_difference* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::set_union

Defined in header hpx/algorithm.hpp[655].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[655] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_union_result<*Iter1*, *Iter2*, *Iter3*>>::type **set_union**(*ExPolicy* &&policy, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in one or both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), then all *m* elements will be copied from [first1, last1) to dest, preserving order, and then exactly std::max(n-m, 0) elements will be copied from [first2, last2) to dest, also preserving order.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_union* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

  ```
  bool pred(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_union* algorithm returns a *hpx::future<ranges::set_union_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_union_result<Iter1, Iter2, Iter3>* otherwise. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, set_union_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::ran

Constructs a sorted range beginning at dest consisting of all elements present in one or both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), then all *m* elements will be copied from [first1, last1) to dest, preserving order, and then exactly std::max(n-m, 0) elements will be copied from [first2, last2) to dest, also preserving order.

The resulting range cannot overlap with either of the input ranges.

The application of function objects in parallel algorithm invoked with a sequential execution policy object execute in sequential order in the calling thread (*sequenced_policy*) or in a single new thread spawned from the current thread (for *sequenced_task_policy*).

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_union* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_union* algorithm returns a *hpx::future<ranges::set_union_result<Iter1, Iter2, Iter3>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::set_union_result<Iter1, Iter2, Iter3>* otherwise. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
set_union_result<*Iter1*, *Iter2*, *Iter3*> **tag_fallback_invoke**(set_union_t, *Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in one or both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), then all *m* elements will be copied from [first1, last1) to dest, preserving order, and then exactly std::max(n-m, 0) elements will be copied from [first2, last2) to dest, also preserving order.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **Iter1** – The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- **Sent2** – The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for Iter2.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_union* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- **last1** – Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **last2** – Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_union* algorithm returns *ranges::set_union_result<Iter1, Iter2, Iter3>*. The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng1**, typename **Rng2**, typename **Iter3**, typename **Pred** = *hpx*::*parallel*::detail::less, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

set_union_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range_iterator_t<*Rng2*>, *Iter3*> **set_union**(*Rng1* &&rng1, *Rng2* &&rng2, *Iter3* dest, *Pred* &&op = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Constructs a sorted range beginning at dest consisting of all elements present in one or both sorted ranges [first1, last1) and [first2, last2). This algorithm expects both input ranges to be sorted with the given binary predicate *f*.

If some element is found *m* times in [first1, last1) and *n* times in [first2, last2), then all *m* elements will be copied from [first1, last1) to dest, preserving order, and then exactly std::max(n-m, 0) elements will be copied from [first2, last2) to dest, also preserving order.

The resulting range cannot overlap with either of the input ranges.

Note: Complexity: At most 2*(N1 + N2 - 1) comparisons, where *N1* is the length of the first sequence and *N2* is the length of the second sequence.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter3** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **Pred** – The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *set_union* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>
- **Proj1** – The type of an optional projection function applied to the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function applied to the second sequence. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **op** – The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type1 &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *InIter* can be dereferenced and then implicitly converted to *Type1*

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *op* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *op* is invoked.

Returns The *set_union* algorithm returns *ranges::set_union_result<Iter1, Iter2, Iter3>*. where Iter1 is range_iterator_t<Rng1> and Iter2 is range_iterator_t<Rng2> The *set_union* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::shift_left

Defined in header hpx/algorithm.hpp[656].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**, typename **Size**>
FwdIter **shift_left**(*FwdIter* first, *Sent* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i.

The assignment operations in the parallel *shift_left* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.

[656] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **n** – Refers to the number of positions to shift.

Returns The *shift_left* algorithm returns *FwdIter*. The *shift_left* algorithm returns an iterator to the end of the resulting range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Size**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **shift_left**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i.

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **n** – Refers to the number of positions to shift.

Returns The *shift_left* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *shift_left* algorithm returns an iterator to the end of the resulting range.

template<typename **Rng**, typename **Size**>
hpx::*traits*::range_iterator_t<*Rng*> **shift_left**(*Rng* &&rng, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i.

The assignment operations in the parallel *shift_left* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Template Parameters

- **Rng** – The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **rng** – Refers to the range in which the elements will be shifted.
- **n** – Refers to the number of positions to shift.

Returns The *shift_left* algorithm returns *hpx::traits::range_iterator_t<Rng>*. The *shift_left* algorithm returns an iterator to the end of the resulting range.

template<typename **ExPolicy**, typename **Rng**, typename **Size**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **shift_left**(*ExPolicy* &&policy, *Rng* &&rng, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the beginning of the range. For every integer i in [0, last - first

- n), moves the element originally at position first + n + i to position first + i.

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignment operations in the parallel *shift_left* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- **Rng** – The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the range in which the elements will be shifted.
- **n** – Refers to the number of positions to shift.

Returns The *shift_left* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::traits::range_iterator_t<Rng>* otherwise. The *shift_left* algorithm returns an iterator to the end of the resulting range.

hpx::ranges::shift_right

Defined in header hpx/algorithm.hpp[657].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**, typename **Size**>
FwdIter **shift_right**(*FwdIter* first, *Sent* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- n + i.

The assignment operations in the parallel *shift_right* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

[657] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **n** – Refers to the number of positions to shift.

Returns The *shift_right* algorithm returns *FwdIter*. The *shift_right* algorithm returns an iterator to the end of the resulting range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Size**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **shift_right**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- n + i.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *FwdIter* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **n** – Refers to the number of positions to shift.

Returns The *shift_right* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *shift_right* algorithm returns an iterator to the end of the resulting range.

template<typename **Rng**, typename **Size**>
hpx::*traits*::range_iterator_t<*Rng*> **shift_right**(*Rng* &&rng, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- n + i.

The assignment operations in the parallel *shift_right* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Template Parameters

- **Rng** – The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **rng** – Refers to the range in which the elements will be shifted.
- **n** – Refers to the number of positions to shift.

Returns The *shift_right* algorithm returns *hpx::traits::range_iterator_t<Rng>*. The *shift_right* algorithm returns an iterator to the end of the resulting range.

template<typename **ExPolicy**, typename **Rng**, typename **Size**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **shift_right**(*ExPolicy* &&policy, *Rng* &&rng, *Size* n)

Shifts the elements in the range [first, last) by n positions towards the end of the range. For every integer i in [0, last - first - n), moves the element originally at position first + i to position first

- n + i.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignment operations in the parallel *shift_right* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: At most (last - first) - n assignments.

Note: The type of dereferenced *hpx::traits::range_iterator_t<Rng>* must meet the requirements of *MoveAssignable*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Size** – The type of the argument specifying the number of positions to shift by.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the range in which the elements will be shifted.
- **n** – Refers to the number of positions to shift.

Returns The *shift_right* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::traits::range_iterator_t<Rng>* otherwise. The *shift_right* algorithm returns an iterator to the end of the resulting range.

hpx::ranges::sort

Defined in header hpx/algorithm.hpp[658].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **RandomIt**, typename **Sent**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
RandomIt **sort**(*RandomIt* first, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(N log(N)), where N = detail::distance(first, last) comparisons.

Template Parameters

- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

[658] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *sort* algorithm returns *RandomIt*. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

template<typename **ExPolicy**, typename **RandomIt**, typename **Sent**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *RandomIt*>::type **sort**(*ExPolicy* &&policy, *RandomIt* first, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = detail::distance(first, last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *sort* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

template<typename **Rng**, typename **Comp**, typename **Proj**>
hpx::*traits*::range_iterator_t<*Rng*> **sort**(*Rng* &&rng, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range *rng* in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(N log(N)), where N = std::distance(begin(rng), end(rng)) comparisons.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *sort* algorithm returns *hpx::traits::range_iterator_t<Rng>*. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **sort**(*ExPolicy* &&policy, *Rng* &&rng, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range *rng* in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = std::distance(begin(rng), end(rng)) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *sort* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::traits::range_iterator_t<Rng>* otherwise. It returns *last*.

hpx::ranges::stable_sort

Defined in header hpx/algorithm.hpp[659].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **RandomIt**, typename **Sent**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
RandomIt **stable_sort**(*RandomIt* first, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *stable_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

[659] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Returns The *stable_sort* algorithm returns *RandomIt*. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

template<typename **ExPolicy**, typename **RandomIt**, typename **Sent**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *RandomIt*>::type **stable_sort**(*ExPolicy* &&policy, *RandomIt* first, *Sent* last, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **RandomIt** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for RandomIt.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *stable_sort* algorithm returns a *hpx::future<RandomIt>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *RandomIt* otherwise.

The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

template<typename **Rng**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
hpx::*traits*::range_iterator_t<*Rng*> **stable_sort**(*Rng* &&rng, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The assignments in the parallel *stable_sort* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *stable_sort* algorithm returns *hpx::traits::range_iterator_t<Rng>*. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **Comp** = *ranges*::less, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*traits*::range_iterator_t<*Rng*>> **stable_sort**(*ExPolicy* &&policy, *Rng* &&rng, *Comp* &&comp = *Comp*(), *Proj* &&proj = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

comp has to induce a strict weak ordering on the values.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The application of function objects in parallel algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(N log(N)), where N = std::distance(first, last) comparisons.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it applies user-provided function objects.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Comp** – The type of the function/function object to use (deduced).
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **comp** – comp is a callable object. The return value of the INVOKE operation applied to an object of type Comp, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.
- **proj** – Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate *comp* is invoked.

Returns The *stable_sort* algorithm returns a *hpx::future<hpx::traits::range_iterator_t<Rng>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *hpx::traits::range_iterator_t<Rng>* otherwise. It returns *last*.

hpx::ranges::starts_with

Defined in header hpx/algorithm.hpp[660].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[660] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **starts_with**(*Iter1* first1, *Sent1* last1, *Iter2* first2, *Sent2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range defined by [first1, last1) matches the prefix of the first range defined by [first2, last2)

The assignments in the parallel *starts_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **Iter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **Iter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a input iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *starts_with* algorithm returns *bool*. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, bool> **starts_with**(*ExPolicy* &&policy, *FwdIter1* first1, *Sent1* last1, *FwdIter2* first2, *Sent2* last2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range defined by [first1, last1) matches the prefix of the first range defined by [first2, last2)

The assignments in the parallel *starts_with* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *starts_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent1** – The type of the end source iterators used(deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- **FwdIter2** – The type of the begin destination iterators used deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end destination iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the source range.
- **last1** – Sentinel value referring to the end of the source range.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Sentinel value referring to the end of the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *starts_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

template<typename **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
bool **starts_with**(*Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range *rng2* matches the prefix of the first range *rng1*.

The assignments in the parallel *starts_with* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The binary predicate that compares the projected elements.
- **Proj1** – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the source range.
- **rng2** – Refers to the destination range.
- **pred** – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *starts_with* algorithm returns *bool*. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, bool>::type **starts_with**(*ExPolicy* &&policy, *Rng1* &&rng1, *Rng2* &&rng2, *Pred* &&pred = *Pred*(), *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Checks whether the second range *rng2* matches the prefix of the first range *rng1*.

The assignments in the parallel *starts_with* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *starts_with* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear: at most min(N1, N2) applications of the predicate and both projections.

Template Parameters

- `ExPolicy` – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1` – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `Rng2` – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `Pred` – The binary predicate that compares the projected elements.
- `Proj1` – The type of an optional projection function for the source range. This defaults to *hpx::identity*
- `Proj2` – The type of an optional projection function for the destination range. This defaults to *hpx::identity*

Parameters

- `policy` – The execution policy to use for the scheduling of the iterations.
- `rng1` – Refers to the source range.
- `rng2` – Refers to the destination range.
- `pred` – Specifies the binary predicate function (or function object) which will be invoked for comparison of the elements in the in two ranges projected by proj1 and proj2 respectively.
- `proj1` – Specifies the function (or function object) which will be invoked for each of the elements in the source range as a projection operation before the actual predicate *is* invoked.
- `proj2` – Specifies the function (or function object) which will be invoked for each of the elements in the destination range as a projection operation before the actual predicate *is* invoked.

Returns The *starts_with* algorithm returns a *hpx::future<bool>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *bool* otherwise. The *starts_with* algorithm returns a boolean with the value true if the second range matches the prefix of the first range, false otherwise.

hpx::ranges::swap_ranges

Defined in header hpx/algorithm.hpp[661].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[661] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **InIter1**, typename **Sent1**, typename **InIter2**, typename **Sent2**>
swap_ranges_result<*InIter1*, *InIter2*> **swap_ranges**(*InIter1* first1, *Sent1* last1, *InIter2* first2, *Sent2* last2)

Exchanges elements between range [first1, last1) and another range starting at *first2*.

The swap operations in the parallel *swap_ranges* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first1* and *last1*

Template Parameters

- **InIter1** – The type of the first range of iterators to swap (deduced).
- **Sent1** – The type of the first sentinel (deduced). This sentinel type must be a sentinel for InIter1.
- **InIter2** – The type of the second range of iterators to swap (deduced).
- **Sent2** – The type of the second sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters

- **first1** – Refers to the beginning of the sequence of elements for the first range.
- **last1** – Refers to sentinel value denoting the end of the sequence of elements for the first range.
- **first2** – Refers to the beginning of the sequence of elements for the second range.
- **last2** – Refers to sentinel value denoting the end of the sequence of elements for the second range.

Returns The *swap_ranges* algorithm returns *swap_ranges_result<InIter1, InIter2>*. The *swap_ranges* algorithm returns in_in_result with the first element as the iterator to the element past the last element exchanged in range beginning with *first1* and the second element as the iterator to the element past the last element exchanged in the range beginning with *first2*.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, swap_ranges_result<*FwdIter1*, *FwdIter2*>>::type **swap_ranges**(*ExPolicy* &&policy, *FwdIter1* first1, *Sent1* last1, *FwdIter2* first2, *Sent2* last2)

Exchanges elements between range [first1, last1) and another range starting at *first2*.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in

unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first1* and *last1*

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the first range of iterators to swap (deduced).
- **Sent1** – The type of the first sentinel (deduced). This sentinel type must be a sentinel for FwdIter1.
- **FwdIter2** – The type of the second range of iterators to swap (deduced).
- **Sent2** – The type of the second sentinel (deduced). This sentinel type must be a sentinel for FwdIter2.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements for the first range.
- **last1** – Refers to sentinel value denoting the end of the sequence of elements for the first range.
- **first2** – Refers to the beginning of the sequence of elements for the second range.
- **last2** – Refers to sentinel value denoting the end of the sequence of elements for the second range.

Returns The *swap_ranges* algorithm returns a *hpx::future<swap_ranges_result<FwdIter1, FwdIter2>>* if the execution policy is of type *parallel_task_policy* and returns *FwdIter2* otherwise. The *swap_ranges* algorithm returns in_in_result with the first element as the iterator to the element past the last element exchanged in range beginning with *first1* and the second element as the iterator to the element past the last element exchanged in the range beginning with *first2*.

template<typename **Rng1**, typename **Rng2**>
swap_ranges_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range_iterator_t<*Rng2*>> **swap_ranges**(*Rng1* &&rng1, *Rng2* &&rng2)

Exchanges elements between range [first1, last1) and another range starting at *first2*.

The swap operations in the parallel *swap_ranges* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first1* and *last1*

Template Parameters
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters
- **rng1** – Refers to the sequence of elements of the first range.
- **rng2** – Refers to the sequence of elements of the second range.

Returns The *swap_ranges* algorithm returns *swap_ranges_result<* hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>. The *swap_ranges* algorithm returns in_in_result with the first element as the iterator to the element past the last element exchanged in range beginning with *first1* and the second element as the iterator to the element past the last element exchanged in the range beginning with *first2*.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, swap_ranges_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range

Exchanges elements between range [first1, last1) and another range starting at *first2*.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The swap operations in the parallel *swap_ranges* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first1* and *last1*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the sequence of elements of the first range.
- **rng2** – Refers to the sequence of elements of the second range.

Returns The *swap_ranges* algorithm returns a *hpx::future<swap_ranges_result<* hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>> if the execution policy is of type *parallel_task_policy* and returns *swap_ranges_result<* hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>. otherwise. The *swap_ranges* algorithm returns in_in_result with the first element as the iterator to the element past the last element exchanged in range beginning with *first1* and the second element as the iterator to the element past the last element exchanged in the range beginning with *first2*.

hpx::ranges::transform

Defined in header hpx/algorithm.hpp[662].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **F**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::unary_transform_result<*FwdIter1*, *FwdIter2*>>::type **transform(**

Applies the given function *f* to the given range *rng* and stores the result in another range, beginning at dest.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly size(rng) applications of *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for FwdIter1.

[662] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *FwdIter2* can be dereferenced and assigned a value of type *Ret*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns a *hpx::future<ranges::unary_transform_result<FwdIter1, FwdIter2> >* if the execution policy is of type *parallel_task_policy* and returns *ranges::unary_transform_result<FwdIter1, FwdIter2>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter**, typename **F**, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::unary_transform_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *FwdIt*

Applies the given function *f* to the given range *rng* and stores the result in another range, beginning at dest.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly size(rng) applications of *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *range_iterator<Rng>::type* can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *OutIter* can be dereferenced and assigned a value of type *Ret*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns a *hpx::future<ranges::unary_transform_result<range_iterator<Rng>::ty* FwdIter>> if the execution policy is of type *parallel_task_policy* and returns *ranges::unary_transform_result<range_iterator<Rng>::type*, FwdIter> otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**, typename **FwdIter3**, typename **F**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::binary_transform_result<*FwdIter1*, *FwdIter2*, *FwdIter3*>>::type t

Applies the given function *f* to pairs of elements from two ranges: one defined by *rng* and the other beginning at first2, and stores the result in another range, beginning at dest.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly size(rng) applications of *f*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for FwdIter1.
- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for FwdIter2.
- **FwdIter3** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.

- **Proj1** – The type of an optional projection function to be used for elements of the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types FwdIter1 and FwdIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter3* can be dereferenced and assigned a value of type *Ret*.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *f* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns A *hpx::future<ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>>* if the execution policy is of type *parallel_task_policy* and returns *ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **FwdIter**, typename **F**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>

parallel::*util*::detail::algorithm_result<*ExPolicy*, *ranges*::binary_transform_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*:

Applies the given function *f* to pairs of elements from two ranges: one defined by [first1, last1) and the other beginning at first2, and stores the result in another range, beginning at dest.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Exactly min(last2-first2, last1-first1) applications of *f*

Note: The algorithm will invoke the binary predicate until it reaches the end of the shorter of the two given input sequences

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj1** – The type of an optional projection function to be used for elements of the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second sequence. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types range_iterator<Rng1>::type and range_iterator<Rng2>::type can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *f* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns a *hpx::future<ranges::binary_transform_result<* hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>, FwdIter> > if the execution policy is of type *parallel_task_policy* and returns *ranges::binary_transform_result<* hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>, FwdIter> otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **F**, typename **Proj** = *hpx*::identity>
ranges::unary_transform_result<*FwdIter1*, *FwdIter2*> **transform**(*FwdIter1* first, *Sent1* last, *FwdIter2* dest, *F* &&f, *Proj* &&proj = *Proj*())

Applies the given function *f* to the given range *rng* and stores the result in another range, beginning at dest.

Note: Complexity: Exactly size(rng) applications of *f*

Template Parameters

- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for FwdIter1.
- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last** – Refers to the end of the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *FwdIter2* can be dereferenced and assigned a value of type *Ret*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns *ranges::unary_transform_result<FwdIter1, FwdIter2>*. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **FwdIter**, typename **F**, typename **Proj** = *hpx*::identity>
ranges::unary_transform_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *FwdIter*> **transform**(*Rng* &&rng, *FwdIter* dest, *F* &&f, *Proj* &&proj = *Proj*())

Applies the given function *f* to the given range *rng* and stores the result in another range, beginning at dest.

Note: Complexity: Exactly size(rng) applications of *f*

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *range_iterator<Rng>::type* can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *OutIter* can be dereferenced and assigned a value of type *Ret*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns *ranges::unary_transform_result<range_iterator<Rng>::type*, FwdIter>. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**, typename **FwdIter3**, typename **F**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
ranges::binary_transform_result<*FwdIter1*, *FwdIter2*, *FwdIter3*> **transform**(*FwdIter1* first1, *Sent1* last1, *FwdIter2* first2, *Sent2* last2, *FwdIter3* dest, *F* &&f, *Proj1* &&proj1 = *Proj1*(), *Proj2* &&proj2 = *Proj2*())

Applies the given function *f* to pairs of elements from two ranges: one defined by *rng* and the other beginning at first2, and stores the result in another range, beginning at dest.

Note: Complexity: Exactly size(rng) applications of *f*

Template Parameters

- **FwdIter1** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent1** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for FwdIter1.
- **FwdIter2** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of a sentinel for FwdIter2.
- **FwdIter3** – The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of a forward iterator.
- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *CopyConstructible*.
- **Proj1** – The type of an optional projection function to be used for elements of the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second sequence. This defaults to *hpx::identity*

Parameters

- **first1** – Refers to the beginning of the first sequence of elements the algorithm will be applied to.
- **last1** – Refers to the end of the first sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- **last2** – Refers to the end of the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types FwdIter1 and FwdIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter3* can be dereferenced and assigned a value of type *Ret*.

- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *f* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns *ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>*. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

template<typename **Rng1**, typename **Rng2**, typename **FwdIter**, typename **F**, typename **Proj1** = *hpx*::identity, typename **Proj2** = *hpx*::identity>
ranges::binary_transform_result<*hpx*::*traits*::range_iterator_t<*Rng1*>, *hpx*::*traits*::range_iterator_t<*Rng2*>, *FwdIter*> **trans**

Applies the given function *f* to pairs of elements from two ranges: one defined by [first1, last1) and the other beginning at first2, and stores the result in another range, beginning at dest.

Note: Complexity: Exactly min(last2-first2, last1-first1) applications of *f*

Note: The algorithm will invoke the binary predicate until it reaches the end of the shorter of the two given input sequences

Template Parameters

- **Rng1** – The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.

- **F** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *transform* requires *F* to meet the requirements of *Copy-Constructible*.
- **Proj1** – The type of an optional projection function to be used for elements of the first sequence. This defaults to *hpx::identity*
- **Proj2** – The type of an optional projection function to be used for elements of the second sequence. This defaults to *hpx::identity*

Parameters

- **rng1** – Refers to the first sequence of elements the algorithm will be applied to.
- **rng2** – Refers to the second sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **f** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types range_iterator<Rng1>::type and range_iterator<Rng2>::type can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter* can be dereferenced and assigned a value of type *Ret*.
- **proj1** – Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *f* is invoked.
- **proj2** – Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *f* is invoked.

Returns The *transform* algorithm returns *ranges::binary_transform_result*< hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>, FwdIter>. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

hpx::ranges::transform_exclusive_scan

Defined in header hpx/algorithm.hpp[663].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

[663] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type>
transform_exclusive_scan_result<*InIter*, *OutIter*> **transform_exclusive_scan**(*InIter* first, *Sent* last, *OutIter* dest, *T* init, *BinOp* &&binary_op, *UnOp* &&unary_op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*(first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN) where 1 < K+1 = M <= N.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_exclusive_scan* algorithm returns *transform_exclusive_scan_result<InIter, OutIter>*. The *transform_exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*FwdIter1*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, transform_exclusive_result<*FwdIter1*, *FwdIter2*>>::type **transform_excl**

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_exclusive_scan* algorithm returns a *hpx::future<transform_exclusive_result<FwdIter1*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_exclusive_result<FwdIter1*, FwdIter2> otherwise. The *transform_exclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **O**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>

transform_exclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*> **transform_exclusive_scan**(*Rng* &&rng, *O* dest, *T* init, *BinOp* &&binary_op, *UnOp* &&unary_op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced).
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_exclusive_scan* algorithm returns a returns *transform_exclusive_scan_result*< traits::range_iterator_t<Rng>, O>. The *transform_exclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **O**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, transform_exclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*>>

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_exclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1

- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator. This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **init** – The initial value for the generalized sum.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_exclusive_scan* algorithm returns a *hpx::future<transform_exclusive_scan_result*< traits::range_iterator_t<Rng>, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_exclusive_scan_result*< traits::range_iterator_t<Rng>, O> otherwise. The *transform_exclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::transform_inclusive_scan

Defined in header hpx/algorithm.hpp[664].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **BinOp**, typename **UnOp**>
transform_inclusive_scan_result<*InIter*, *OutIter*> **transform_inclusive_scan**(*InIter* first, *Sent* last, *OutIter* dest, *BinOp* &&binary_op, *UnOp* &&unary_op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.

[664] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_inclusive_scan* algorithm returns *transform_inclusive_scan_result<InIter, OutIter>*. The *transform_inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **BinOp**, typename **UnOp**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, transform_inclusive_result<*FwdIter1*, *FwdIter2*>>::type **transform_incl**

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_inclusive_scan* algorithm returns a *hpx::future<transform_inclusive_result<FwdIter1*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_inclusive_result<FwdIter1*, FwdIter2> otherwise. The *transform_inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **O**, typename **BinOp**, typename **UnOp**>
transform_inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*> **transform_inclusive_scan**(*Rng* &&rng, *O* dest, *BinOp* &&binary_op, *UnOp* &&unary_op)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.

- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_inclusive_scan* algorithm returns a returns *transform_inclusive_scan_result*< traits::range_iterator_t<Rng>, O>. The *transform_inclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **O**, typename **BinOp**, typename **UnOp**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, transform_inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*>>:

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator. This iterator type must meet the requirements of an forward iterator.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_inclusive_scan* algorithm returns a *hpx::future*<*transform_inclusive_scan_result*< traits::range_iterator_t<Rng>, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_inclusive_scan_result*< traits::range_iterator_t<Rng>, O> otherwise. The *transform_inclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **Sent**, typename **OutIter**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*InIter*>::value_type>
transform_inclusive_scan_result<*InIter*, *OutIter*> **transform_inclusive_scan**(*InIter* first, *Sent* last, *OutIter* dest, *BinOp* &&binary_op, *UnOp* &&unary_op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN) where 1 < K+1 = M <= N.

Template Parameters
- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **OutIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.
- **init** – The initial value for the generalized sum.

Returns The *transform_inclusive_scan* algorithm returns *transform_inclusive_scan_result<InIter, OutIter>*. The *transform_inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent**, typename **FwdIter2**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*FwdIter1*>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, transform_inclusive_result<*FwdIter1*, *FwdIter2*>>::type **transform_incl**

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.
- **init** – The initial value for the generalized sum.

Returns The *transform_inclusive_scan* algorithm returns a *hpx::future<transform_inclusive_result<FwdIter1*, FwdIter2>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_inclusive_result<FwdIter1*, FwdIter2> otherwise. The *transform_inclusive_scan* algorithm returns an input iterator to the point denoted by the sentinel and an output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng**, typename **O**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
transform_inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*> **transform_inclusive_scan**(*Rng* &&rng, *O* dest, *BinOp* &&binary_op, *UnOp* &&unary_op, *T* init)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:
- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **O** – The type of the iterator representing the destination range (deduced).
- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.
- **init** – The initial value for the generalized sum.

Returns The *transform_inclusive_scan* algorithm returns a returns *transform_inclusive_scan_result*< traits::range_iterator_t<Rng>, O>. The *trans-*

form_inclusive_scan algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **O**, typename **BinOp**, typename **UnOp**, typename **T** = typename *std*::iterator_traits<*hpx*::*traits*::range_iterator_t<*Rng*>>::value_type>
parallel::*util*::detail::algorithm_result<*ExPolicy*, transform_inclusive_scan_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*>>:

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result)))).

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_inclusive_scan* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither *conv* nor *op* shall invalidate iterators or sub-ranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *op* and *conv*.

Note: GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aN) is defined as:

- a1 when N is 1
- op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, …, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, …, aN) where 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator. This iterator type must meet the requirements of an forward iterator.

- **BinOp** – The type of the binary function object used for the reduction operation.
- **UnOp** – The type of the unary function object used for the conversion operation.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **binary_op** – Specifies the function (or function object) which will be invoked for each of the values of the input sequence. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Ret* must be such that an object of a type as given by the input sequence can be implicitly converted to any of those types.
- **unary_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.
- **init** – The initial value for the generalized sum.

Returns The *transform_inclusive_scan* algorithm returns a *hpx::future<transform_inclusive_scan_result*< traits::range_iterator_t<Rng>, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *transform_inclusive_scan_result*< traits::range_iterator_t<Rng>, O> otherwise. The *transform_inclusive_scan* algorithm returns an input iterator to one past the end of the range and an output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::transform_reduce

Defined in header hpx/algorithm.hpp[665].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[665] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Functions

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **T**, typename **Reduce**, typename **Convert**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.
- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *parallel_task_policy* and returns *T* otherwise. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **Iter**, typename **Sent**, typename **T**, typename **Reduce**, typename **Convert**>
T **transform_reduce**(*Iter* first, *Sent* last, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.

- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns *T*. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **Iter2**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *Iter* first, *Sent* last, *Iter2* first2, *T* init)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an random access iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **init** – The initial value for the generalized sum.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *parallel_task_policy* and returns *T* otherwise. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **Iter**, typename **Sent**, typename **Iter2**, typename **T**>
T **transform_reduce**(*Iter* first, *Sent* last, *Iter2* first2, *T* init)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- `Iter` – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- `Sent` – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- `Iter2` – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an random access iterator.
- `T` – The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `first` – Refers to the beginning of the first sorted range the algorithm will be applied to.
- `last` – Refers to the end of the second sorted range the algorithm will be applied to.
- `first2` – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `init` – The initial value for the generalized sum.

Returns The *transform_reduce* algorithm returns *T*. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

```
template<typename ExPolicy, typename Iter, typename Sent, typename Iter2, typename T, typename Reduce, typename Convert>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, T> transform_reduce(ExPolicy &&policy, Iter first, Sent last, Iter2 first2, T init, Reduce &&red_op, Convert &&conv_op)
```

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an random access iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.
- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.
- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *parallel_task_policy* and returns *T* otherwise. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **Iter**, typename **Sent**, typename **Iter2**, typename **T**, typename **Reduce**, typename **Convert**>
T **transform_reduce**(*Iter* first, *Sent* last, *Iter2* first2, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **Iter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an random access iterator.
- **Sent** – The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- **Iter2** – The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an random access iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **first** – Refers to the beginning of the first sorted range the algorithm will be applied to.
- **last** – Refers to the end of the second sorted range the algorithm will be applied to.

- **first2** – Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.
- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns *T*. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **Rng**, typename **T**, typename **Reduce**, typename **Convert**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *Rng* &&rng, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The reduce operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:

- b1, …, bN may be any permutation of a1, …, aN and
- 1 < K+1 = M <= N.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.
- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *parallel_task_policy* and returns *T* otherwise. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **Rng**, typename **T**, typename **Reduce**, typename **Convert**>
T **transform_reduce**(*Rng* &&rng, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns GENERALIZED_SUM(red_op, init, conv_op(*first), …, conv_op(*(first + (last - first) - 1))).

The difference between *transform_reduce* and *accumulate* is that the behavior of transform_reduce may be non-deterministic for non-associative or non-commutative binary predicate.

Note: Complexity: O(*last* - *first*) applications of the predicates *red_op* and *conv_op*.

Note: GENERALIZED_SUM(op, a1, …, aN) is defined as follows:

- a1 when N is 1
- op(GENERALIZED_SUM(op, b1, …, bK), GENERALIZED_SUM(op, bM, …, bN)), where:
  - b1, …, bN may be any permutation of a1, …, aN and
  - 1 < K+1 = M <= N.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be used as initial (and intermediate) values (deduced).
- **Reduce** – The type of the binary function object used for the reduction operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **init** – The initial value for the generalized sum.
- **red_op** – Specifies the function (or function object) which will be invoked for each of the values returned from the invocation of *conv_op*. This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

Returns The *transform_reduce* algorithm returns *T*. The *transform_reduce* algorithm returns the result of the generalized sum over the values returned from *conv_op* when applied to the elements given by the input range [first, last).

template<typename **ExPolicy**, typename **Rng**, typename **Iter2**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *Rng* &&rng, *Iter2* first2, *T* init)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(*last* - *first*) applications of the predicate *op2*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as return) values (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise.

template<typename **Rng**, typename **Iter2**, typename **T**>
T **transform_reduce**(*Rng* &&rng, *Iter2* first2, *T* init)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

Note: Complexity: O(*last* - *first*) applications of the predicate *op2*.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

- **Iter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as return) values (deduced).

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.

Returns The *transform_reduce* algorithm returns *T*.

template<typename **ExPolicy**, typename **Rng**, typename **Iter2**, typename **T**, typename **Reduce**, typename **Convert**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *T*> **transform_reduce**(*ExPolicy* &&policy, *Rng* &&rng, *Iter2* first2, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The operations in the parallel *transform_reduce* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: O(*last* - *first*) applications of the predicate *op2*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as return) values (deduced).
- **Reduce** – The type of the binary function object used for the multiplication operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.

- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.
- **red_op** – Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *op2*. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.
- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to an object for the second argument type of *op1*.

Returns The *transform_reduce* algorithm returns a *hpx::future<T>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *T* otherwise.

template<typename **Rng**, typename **Iter2**, typename **T**, typename **Reduce**, typename **Convert**>
T **transform_reduce**(*Rng* &&rng, *Iter2* first2, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

Note: Complexity: O(*last* - *first*) applications of the predicate *op2*.

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Iter2** – The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T** – The type of the value to be used as return) values (deduced).
- **Reduce** – The type of the binary function object used for the multiplication operation.
- **Convert** – The type of the unary function object used to transform the elements of the input sequence before invoking the reduce function.

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **first2** – Refers to the beginning of the second sequence of elements the result will be calculated with.
- **init** – The initial value for the sum.

- **red_op** – Specifies the function (or function object) which will be invoked for the initial value and each of the return values of *op2*. This is a binary predicate. The signature of this predicate should be equivalent to should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.

- **conv_op** – Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to an object for the second argument type of *op1*.

Returns The *transform_reduce* algorithm returns *T*.

hpx::ranges::uninitialized_copy, hpx::ranges::uninitialized_copy_n

Defined in header hpx/algorithm.hpp[666].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter**, typename **Sent1**, typename **FwdIter**, typename **Sent2**>
hpx::*parallel*::*util*::in_out_result<*InIter*, *FwdIter*> **uninitialized_copy**(*InIter* first1, *Sent1* last1, *FwdIter* first2, *Sent2* last2)

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.

[666] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters

- **first1** – Refers to the beginning of the sequence of elements that will be copied from
- **last1** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns The *uninitialized_copy* algorithm returns an *in_out_result<InIter, FwdIter>*. The *uninitialized_copy* algorithm returns an input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *parallel*::*util*::in_out_result<*FwdIter1*, *FwdIter2*>>::type **uninitialized**

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements that will be copied from
- **last1** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns The *uninitialized_copy* algorithm returns a *hpx::future<in_out_result<InIter, FwdIter>>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<InIter, FwdIter>* otherwise. The *uninitialized_copy* algorithm returns an input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

template<typename **Rng1**, typename **Rng2**>
hpx::*parallel*::*util*::in_out_result<typename *hpx*::*traits*::range_traits<*Rng1*>::iterator_type, typename *hpx*::*traits*::range_trait

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- **rng1** – Refers to the range from which the elements will be copied from
- **rng2** – Refers to the range to which the elements will be copied to

Returns The *uninitialized_copy* algorithm returns an *in_out_result<typename* hpx::traits::range_traits<Rng1>::iterator_type, typename hpx::traits::range_traits<Rng2>::iterator_type>. The *uninitialized_copy* algorithm returns an input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*parallel*::*util*::in_out_result<typename *hpx*::*traits*::range_traits<*Rng1*

Copies the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the range from which the elements will be copied from
- **rng2** – Refers to the range to which the elements will be copied to

Returns The *uninitialized_copy* algorithm returns a *hpx::future<in_out_result<InIter, FwdIter>>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<* typename hpx::traits::range_traits<Rng1>::iterator_type , typename hpx::traits::range_traits<Rng2>::iterator_type> otherwise. The *uninitialized_copy* algorithm returns the input iterator to one past the last element copied from and the output iterator to the element in the destination range, one past the last element copied.

template<typename **InIter**, typename **Size**, typename **FwdIter**, typename **Sent2**>
hpx::*parallel*::*util*::in_out_result<*InIter*, *FwdIter*> **uninitialized_copy_n**(*InIter* first1, *Size* count, *FwdIter* first2, *Sent2* last2)

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last - first* assignments.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters

- **first1** – Refers to the beginning of the sequence of elements that will be copied from

- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns The *uninitialized_copy_n* algorithm returns *in_out_result<InIter, FwdIter>*. The *uninitialized_copy_n* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**, typename **Sent2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *parallel*::*util*::in_out_result<*FwdIter1*, *FwdIter2*>>::type **uninitialized**

Copies the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the copy operation, the function has no effects.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_copy_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements that will be copied from
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **first2** – Refers to the beginning of the destination range.

- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns The *uninitialized_copy_n* algorithm returns a *hpx::future<in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_copy_n* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

hpx::ranges::uninitialized_default_construct, hpx::ranges::uninitialized_default_construct_n

Defined in header hpx/algorithm.hpp[667].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**>
FwdIter **uninitialized_default_construct**(*FwdIter* first, *Sent* last)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

Returns The *uninitialized_default_construct* algorithm returns a returns *FwdIter*. The *uninitialized_default_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**>

[667] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_default_construct**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

Returns The *uninitialized_default_construct* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_default_construct* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **Rng**>
hpx::*traits*::range_traits<*Rng*>::iterator_type **uninitialized_default_construct**(*Rng* &&rng)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters **rng** – Refers to the range to which will be default constructed.

Returns The *uninitialized_default_construct* algorithm returns a returns *hpx::traits::range_traits<Rng>::iterator_type*. The *uninitialized_default_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

template<typename **ExPolicy**, typename **Rng**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::type **uninitial**

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_default_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the range to which the value will be default constructed

Returns The *uninitialized_default_construct* algorithm returns a *hpx::future<typename* hpx::traits::range_traits<Rng>::iterator_type>, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *typename* hpx::traits::range_traits<Rng>::iterator_type otherwise. The *uninitialized_default_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

template<typename **FwdIter**, typename **Size**>
FwdIter **uninitialized_default_construct_n**(*FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range [first, first + count) by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct_n* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_default_construct_n* algorithm returns a returns *FwdIter*. The *uninitialized_default_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **uninitialized_default_construct_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range [first, first + count) by default-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_default_construct_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_default_construct_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_default_construct_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns

FwdIter otherwise. The *uninitialized_default_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

hpx::ranges::uninitialized_fill, hpx::ranges::uninitialized_fill_n

Defined in header hpx/algorithm.hpp[668].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**, typename **T**>
FwdIter **uninitialized_fill**(*FwdIter* first, *Sent* last, *T* const &value)

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

The assignments in the ranges *uninitialized_fill* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **value** – The value to be assigned.

Returns The *uninitialized_fill* algorithm returns a returns *FwdIter*. The *uninitialized_fill* algorithm returns the output iterator to the element in the range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_fill**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *T* const &value)

[668] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **T** – The type of the value to be assigned (deduced).

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **value** – The value to be assigned.

Returns The *uninitialized_fill* algorithm returns a returns *FwdIter*. The *uninitialized_fill* algorithm returns the output iterator to the element in the range, one past the last element copied.

template<typename **Rng**, typename **T**>
hpx::*traits*::range_traits<*Rng*>::iterator_type **uninitialized_fill**(*Rng* &&rng, *T* const &value)

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Linear in the distance between *first* and *last*

Template Parameters
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters
- **rng** – Refers to the range to which the value will be filled
- **value** – The value to be assigned.

Returns The *uninitialized_fill* algorithm returns a returns *hpx::traits::range_traits<Rng>::iterator_type*. The *uninitialized_fill* algorithm returns the output iterator to the element in the range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **T**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, typename *hpx*::*traits*::range_traits<Rng1>::iterator_type>::type **uninitia**

Copies the given *value* to an uninitialized memory area, defined by the range [first, last). If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_fill* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Linear in the distance between *first* and *last*

Template Parameters
- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **T** – The type of the value to be assigned (deduced).

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the range to which the value will be filled
- **value** – The value to be assigned.

Returns The *uninitialized_fill* algorithm returns a *hpx::future<typename* hpx::traits::range_traits<Rng>::iterator_type>, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *typename* hpx::traits::range_traits<Rng>::iterator_type otherwise. The *uninitialized_fill* algorithm returns the iterator to one past the last element filled in the range.

template<typename **FwdIter**, typename **Size**, typename **T**>
FwdIter **uninitialized_fill_n**(*FwdIter* first, *Size* count, *T* const &value)

Copies the given *value* value to the first count elements in an uninitialized memory area beginning at first. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **value** – The value to be assigned.

Returns The *uninitialized_fill_n* algorithm returns a returns *FwdIter*. The *uninitialized_fill_n* algorithm returns the output iterator to the element in the range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**, typename **T**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_fill_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count, *T* const &value)

Copies the given *value* value to the first count elements in an uninitialized memory area beginning at first. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **T** – The type of the value to be assigned (deduced).

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

- **value** – The value to be assigned.

Returns The *uninitialized_fill_n* algorithm returns a *hpx::future<FwdIter>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns FwdIter otherwise. The *uninitialized_fill_n* algorithm returns the output iterator to the element in the range, one past the last element copied.

hpx::ranges::uninitialized_move, hpx::ranges::uninitialized_move_n

Defined in header hpx/algorithm.hpp[669].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **InIter**, typename **Sent1**, typename **FwdIter**, typename **Sent2**>
hpx::*parallel*::*util*::in_out_result<*InIter*, *FwdIter*> **uninitialized_move**(*InIter* first1, *Sent1* last1, *FwdIter* first2, *Sent2* last2)

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters
- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters
- **first1** – Refers to the beginning of the sequence of elements that will be moved from
- **last1** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

[669] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

Returns The *uninitialized_move* algorithm returns an *in_out_result<InIter, FwdIter>*. The *uninitialized_move* algorithm returns an input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter2**, typename **Sent2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *parallel*::*util*::in_out_result<*FwdIter1*, *FwdIter2*>>::type **uninitialized**

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent1** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements that will be moved from
- **last1** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns The *uninitialized_move* algorithm returns a *hpx::future<in_out_result<InIter, FwdIter>>*, if the execution policy is of type *sequenced_task_policy* or *paral-*

lel_task_policy and returns *in_out_result<InIter, FwdIter>* otherwise. The *uninitialized_move* algorithm returns an input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

template<typename **Rng1**, typename **Rng2**>
hpx::*parallel*::*util*::in_out_result<typename *hpx*::*traits*::range_traits<*Rng1*>::iterator_type, typename *hpx*::*traits*::range_trait

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- **rng1** – Refers to the range from which the elements will be moved from
- **rng2** – Refers to the range to which the elements will be moved to

Returns The *uninitialized_move* algorithm returns an *in_out_result<typename* hpx::traits::range_traits<Rng1>::iterator_type, typename hpx::traits::range_traits<Rng2>::iterator_type>. The *uninitialized_move* algorithm returns an input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*parallel*::*util*::in_out_result<typename *hpx*::*traits*::range_traits<*Rng1*:

Moves the elements in the range, defined by [first, last), to an uninitialized memory area beginning at *dest*. If an exception is thrown during the initialization, some objects in [first, last) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_move* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng1** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- **Rng2** – The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng1** – Refers to the range from which the elements will be moved from
- **rng2** – Refers to the range to which the elements will be moved to

Returns The *uninitialized_move* algorithm returns a *hpx::future<in_out_result<InIter, FwdIter>>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result*< typename hpx::traits::range_traits<Rng1>::iterator_type , typename hpx::traits::range_traits<Rng2>::iterator_type> otherwise. The *uninitialized_move* algorithm returns the input iterator to one past the last element moved from and the output iterator to the element in the destination range, one past the last element moved.

template<typename **InIter**, typename **Size**, typename **FwdIter**, typename **Sent2**>
hpx::*parallel*::*util*::in_out_result<*InIter*, *FwdIter*> **uninitialized_move_n**(*InIter* first1, *Size* count, *FwdIter* first2, *Sent2* last2)

Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *count* movements, if count > 0, no move operations otherwise.

Template Parameters

- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters

- **first1** – Refers to the beginning of the sequence of elements that will be moved from
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns The *uninitialized_move_n* algorithm returns *in_out_result<InIter, FwdIter>*. The *uninitialized_move_n* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**, typename **Sent2**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *parallel*::*util*::in_out_result<*FwdIter1*, *FwdIter2*>>::type **uninitialized**

Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_move_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* movements, if count > 0, no move operations otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter1** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.
- **FwdIter2** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent2** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter2.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first1** – Refers to the beginning of the sequence of elements that will be moved from
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.
- **first2** – Refers to the beginning of the destination range.
- **last2** – Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

Returns The *uninitialized_move_n* algorithm returns a *hpx::future<in_out_result<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *uninitialized_move_n* algorithm returns the output iterator to the element in the destination range, one past the last element moved.

hpx::ranges::uninitialized_value_construct, hpx::ranges::uninitialized_value_construct_n

Defined in header hpx/algorithm.hpp[670].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**>
FwdIter **uninitialized_value_construct**(*FwdIter* first, *Sent* last)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

Returns The *uninitialized_value_construct* algorithm returns a returns *FwdIter*. The *uninitialized_value_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_value_construct**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

[670] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.

Returns The *uninitialized_value_construct* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_value_construct* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **Rng**>
hpx::*traits*::range_traits<*Rng*>::iterator_type **uninitialized_value_construct**(*Rng* &&rng)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters **rng** – Refers to the range to which will be value constructed.

Returns The *uninitialized_value_construct* algorithm returns a returns *hpx::traits::range_traits<Rng>::iterator_type*. The *uninitialized_value_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

template<typename **ExPolicy**, typename **Rng**>

parallel::*util*::detail::algorithm_result<*ExPolicy*, typename *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::type **uninitial**

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_value_construct* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *last* - *first* assignments.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the range to which the value will be value consutrcted

Returns The *uninitialized_value_construct* algorithm returns a *hpx::future<typename* hpx::traits::range_traits<Rng>::iterator_type>, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *typename* hpx::traits::range_traits<Rng>::iterator_type otherwise. The *uninitialized_value_construct* algorithm returns the output iterator to the element in the range, one past the last element constructed.

template<typename **FwdIter**, typename **Size**>
FwdIter **uninitialized_value_construct_n**(*FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct_n* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_value_construct_n* algorithm returns a returns *FwdIter*. The *uninitialized_value_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter*> **uninitialized_value_construct_n**(*ExPolicy* &&policy, *FwdIter* first, *Size* count)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range [first, first + count) by value-initialization. If an exception is thrown during the initialization, the function has no effects.

The assignments in the parallel *uninitialized_value_construct_n* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *uninitialized_value_construct_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs exactly *count* assignments, if count > 0, no assignments otherwise.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Size** – The type of the argument specifying the number of elements to apply *f* to.

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **count** – Refers to the number of elements starting at *first* the algorithm will be applied to.

Returns The *uninitialized_value_construct_n* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *uninitialized_value_construct_n* algorithm returns the iterator to the element in the source range, one past the last element constructed.

hpx::ranges::unique, hpx::ranges::unique_copy

Defined in header hpx/algorithm.hpp[671].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **ranges**

Functions

template<typename **FwdIter**, typename **Sent**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
subrange_t<*FwdIter*, *Sent*> **unique**(*FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

[671] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/algorithm.hpp

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique* algorithm returns *subrange_t<FwdIter, Sent>*. The *unique* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*FwdIter*, *Sent*>>::type **unique**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*.

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique* algorithm returns *subrange_t<FwdIter, Sent>*. The *unique* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange.

template<typename **Rng**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>, *hpx*::*traits*::range_iterator_t<*Rng*>> **unique**(*Rng* &&rng, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Eliminates all but the first element from every consecutive group of equivalent elements from the range *rng* and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Note: Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique* algorithm returns *subrange_t<hpx::traits::range_iterator_t<Rng>*, hpx::traits::range_iterator_t<Rng>>. The *unique* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange.

template<typename **ExPolicy**, typename **Rng**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, subrange_t<*hpx*::*traits*::range_iterator_t<*Rng*>, *hpx*::*traits*::range_iterator_

Eliminates all but the first element from every consecutive group of equivalent elements from the range *rng* and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be

dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique* algorithm returns a *hpx::future* <subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *subrange_t<hpx::traits::range_iterator_t<Rng>,* hpx::traits::range_iterator_t<Rng>> otherwise. The *unique* algorithm returns an object {ret, last}, where ret is a past-the-end iterator for a new subrange.

template<typename **InIter**, typename **Sent**, typename **O**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
unique_copy_result<*InIter*, *O*> **unique_copy**(*InIter* first, *Sent* last, *O* dest, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*

Template Parameters
- **InIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for InIter.
- **O** – The type of the iterator representing the destination range (deduced).
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter1* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique_copy* algorithm returns a returns unique_copy_result<InIter, O>. The *unique_copy* algorithm returns an in_out_result with the source iterator to one past the last element and out containing the destination iterator to the end of the *dest* range.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **O**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, unique_copy_result<*FwdIter*, *O*>>::type **unique_copy**(*ExPolicy* &&policy, *FwdIter* first, *Sent* last, *O* dest, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than *last* - *first* assignments, exactly *last* - *first* - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **FwdIter** – The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- **Sent** – The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter1.
- **O** – The type of the iterator representing the destination range (deduced).
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **policy** – The execution policy to use for the scheduling of the iterations.

- **first** – Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last** – Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique_copy* algorithm returns returns a hpx::future< unique_copy_result<FwdIter, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *unique_copy_result<FwdIter, O>* otherwise. The *unique_copy* algorithm returns an in_out_result with the source iterator to one past the last element and out containing the destination iterator to the end of the *dest* range.

template<typename **Rng**, typename **O**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
unique_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*> **unique_copy**(*Rng* &&rng, *O* dest, *Pred* &&pred = *Pred*(), *Proj* &&proj = *Proj*())

Copies the elements from the range *rng*, to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note: Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters

- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be

equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique_copy* algorithm returns *unique_copy_result*< hpx::traits::range_iterator_t<Rng>, O>. The *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

template<typename **ExPolicy**, typename **Rng**, typename **O**, typename **Pred** = *ranges*::equal_to, typename **Proj** = *hpx*::identity>
parallel::*util*::detail::algorithm_result<*ExPolicy*, unique_copy_result<*hpx*::*traits*::range_iterator_t<*Rng*>, *O*>> **unique_cop**

Copies the elements from the range *rng*, to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Note: Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- **ExPolicy** – The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- **Rng** – The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- **O** – The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.
- **Pred** – The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique_copy* requires *Pred* to meet the requirements of

CopyConstructible. This defaults to std::equal_to<>
- **Proj** – The type of an optional projection function. This defaults to *hpx::identity*

Parameters
- **policy** – The execution policy to use for the scheduling of the iterations.
- **rng** – Refers to the sequence of elements the algorithm will be applied to.
- **dest** – Refers to the beginning of the destination range.
- **pred** – Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.
- **proj** – Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

Returns The *unique_copy* algorithm returns a *hpx::future<unique_copy_result<* hpx::traits::range_iterator_t<Rng>, O>> if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *unique_copy_result<* hpx::traits::range_iterator_t<Rng>, O> otherwise. The *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

hpx/parallel/util/range.hpp

Defined in header hpx/parallel/util/range.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **util**

Typedefs

template<typename **Iterator**, typename **Sentinel** = *Iterator*>

using **range** = *hpx*::*util*::iterator_range<*Iterator*, *Sentinel*>

Functions

template<typename **Iter**, typename **Sent**>
range<*Iter*, *Sent*> **concat**(*range*<*Iter*, *Sent*> const &it1, *range*<*Iter*, *Sent*> const &it2)

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**>
range<*Iter2*, *Iter2*> **init_move**(*range*<*Iter2*, *Sent2*> const &dest, *range*<*Iter1*, *Sent1*> const &src)

Move objects from the range src to dest.

Parameters
- **dest** – **[in]** : range where move the objects
- **src** – **[in]** : range from where move the objects

Returns range with the objects moved and the size adjusted

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**>
range<*Iter2*, *Sent2*> **uninit_move**(*range*<*Iter2*, *Sent2*> const &dest, *range*<*Iter1*, *Sent1*> const &src)

Move objects from the range src creating them in dest.

Parameters
- **dest** – **[in]** : range where move and create the objects
- **src** – **[in]** : range from where move the objects

Returns range with the objects moved and the size adjusted

template<typename **Iter**, typename **Sent**>
void **destroy_range**(*range*<*Iter*, *Sent*> r)

destroy a range of objects

Parameters **r** – **[in]** : range to destroy

template<typename **Iter**, typename **Sent**>
range<*Iter*, *Sent*> **init**(*range*<*Iter*, *Sent*> const &r, typename *std*::iterator_traits<*Iter*>::value_type &val)

initialize a range of objects with the object val moving across them

Parameters
- **r** – **[in]** : range of elements not initialized
- **val** – **[in]** : object used for the initialization

Returns range initialized

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Compare**>
bool **is_mergeable**(*range*<*Iter1*, *Sent1*> const &src1, *range*<*Iter2*, *Sent2*> const &src2, *Compare* comp)

: indicate if two ranges have a possible merge

Remark

Parameters
- **src1** – **[in]** : first range
- **src2** – **[in]** : second range
- **comp** – **[in]** : object for to compare elements

Returns true : they can be merged false : they can't be merged

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Sent3**, typename **Compare**>

range<*Iter3*, *Sent3*> **full_merge**(*range*<*Iter3*, *Sent3*> const &dest, *range*<*Iter1*, *Sent1*> const &src1, *range*<*Iter2*, *Sent2*> const &src2, *Compare* comp)

Merge two contiguous ranges src1 and src2 , and put the result in the range dest, returning the range merged.

Parameters

- **dest** – **[in]** : range where locate the elements merged. the size of dest must be greater or equal than the sum of the sizes of src1 and src2
- **src1** – **[in]** : first range to merge
- **src2** – **[in]** : second range to merge
- **comp** – **[in]** : comparison object

Returns range with the elements merged and the size adjusted

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Value**, typename **Compare**>
range<*Value**> **uninit_full_merge**(*range*<*Value**> const &dest, *range*<*Iter1*, *Sent1*> const &src1, *range*<*Iter2*, *Sent2*> const &src2, *Compare* comp)

Merge two contiguous ranges src1 and src2 , and create and move the result in the uninitialized range dest, returning the range merged.

Parameters

- **dest** – **[in]** : range where locate the elements merged. the size of dest must be greater or equal than the sum of the sizes of src1 and src2. Initially is un-initialize memory
- **src1** – **[in]** : first range to merge
- **src2** – **[in]** : second range to merge
- **comp** – **[in]** : comparison object

Returns range with the elements merged and the size adjusted

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Compare**>
range<*Iter2*, *Sent2*> **half_merge**(*range*<*Iter2*, *Sent2*> const &dest, *range*<*Iter1*, *Sent1*> const &src1, *range*<*Iter2*, *Sent2*> const &src2, *Compare* comp)

: Merge two buffers. The first buffer is in a separate memory

Parameters

- **dest** – **[in]** : range where finish the two buffers merged
- **src1** – **[in]** : first range to merge in a separate memory
- **src2** – **[in]** : second range to merge, in the final part of the range where deposit the final results
- **comp** – **[in]** : object for compare two elements of the type pointed by the Iter1 and Iter2

Returns : range with the two buffers merged

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Iter3**, typename **Sent3**, typename **Compare**>
bool **in_place_merge_uncontiguous**(*range*<*Iter1*, *Sent1*> const &src1, *range*<*Iter2*, *Sent2*> const &src2, *range*<*Iter3*, *Sent3*> &aux, *Compare* comp)

: merge two non contiguous buffers src1 , src2, using the range aux as auxiliary memory

---

Remark

Parameters

- **src1** – **[in]** : first range to merge
- **src2** – **[in]** : second range to merge

- **aux** – **[in]** : auxiliary range used in the merge
- **comp** – **[in]** : object for to compare elements

Returns true : not changes done false : changes in the buffers

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Compare**>
range<*Iter1*, *Sent1*> **in_place_merge**(*range*<*Iter1*, *Sent1*> const &src1, *range*<*Iter1*, *Sent1*> const &src2, *range*<*Iter2*, *Sent2*> &buf, *Compare* comp)

: merge two contiguous buffers (src1, src2) using buf as auxiliary memory

---

Remark

Parameters

- **src1** – **[in]** : first range to merge
- **src2** – **[in]** : second range to merge
- **buf** – **[in]** : auxiliary memory used in the merge
- **comp** – **[in]** : object for to compare elements

Returns true : not changes done false : changes in the buffers

template<typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Compare**>
void **merge_flow**(*range*<*Iter1*, *Sent1*> rng1, *range*<*Iter2*, *Sent2*> rbuf, *range*<*Iter1*, *Sent1*> rng2, *Compare* cmp)

asio

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/asio/asio_util.hpp

Defined in header hpx/asio/asio_util.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

Typedefs

using **endpoint_iterator_type** = asio::ip::tcp::resolver::iterator

Functions

bool **get_endpoint**(*std*::string const &addr, *std*::uint16_t port, asio::ip::tcp::endpoint &ep, bool force_ipv4 = false)

std::string **get_endpoint_name**(asio::ip::tcp::endpoint const &ep)

asio::ip::tcp::endpoint **resolve_hostname**(*std*::string const &hostname, *std*::uint16_t port, asio::io_context &io_service, bool force_ipv4 = false)

std::string **resolve_public_ip_address**()

std::string **cleanup_ip_address**(*std*::string const &addr)

endpoint_iterator_type **connect_begin**(*std*::string const &address, *std*::uint16_t port, asio::io_context &io_service)

template<typename **Locality**>
endpoint_iterator_type **connect_begin**(*Locality* const &loc, asio::io_context &io_service)

Returns an iterator which when dereferenced will give an endpoint suitable for a call to connect() related to this locality.

inline *endpoint_iterator_type* **connect_end**()

endpoint_iterator_type **accept_begin**(*std*::string const &address, *std*::uint16_t port, asio::io_context &io_service)

template<typename **Locality**>
endpoint_iterator_type **accept_begin**(*Locality* const &loc, asio::io_context &io_service)

Returns an iterator which when dereferenced will give an endpoint suitable for a call to accept() related to this locality.

inline *endpoint_iterator_type* **accept_end**()

bool **split_ip_address**(*std*::string const &v, *std*::string &host, *std*::uint16_t &port)

assertion

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/assertion/evaluate_assert.hpp

Defined in header hpx/assertion/evaluate_assert.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **assertion**

HPX_CURRENT_SOURCE_LOCATION, hpx::source_location

Defined in header hpx/source_location.hpp[672].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_CURRENT_SOURCE_LOCATION()

namespace **hpx**

Functions

std::ostream &**operator<<**(*std*::ostream &os, *source_location* const &loc)

struct **source_location**

#include <source_location.hpp> This contains the location information where *HPX_ASSERT* has been called The source_location class represents certain information about the source code, such as file names, line numbers, and function names. Previously, functions that desire to obtain this information about the call site (for logging, testing, or debugging purposes) must use macros so that predefined macros like and are expanded in the context of the caller. The source_location class provides a better alternative. source_location meets the *DefaultConstructible*, *CopyConstructible*, *CopyAssignable* and *Destructible* requirements. Lvalue of source_location meets the Swappable requirement. Additionally, the following conditions are true:

- ```
  std::is_nothrow_move_constructible_v<std::source_location>
  ```

- ```
  std::is_nothrow_move_assignable_v<std::source_location>
  ```

- ```
  std::is_nothrow_swappable_v<std::source_location>
  ```

It is intended that *source_location* has a small size and can be copied efficiently. It is unspecified whether the copy/move constructors and the copy/move assignment operators of source_location are trivial and/or constexpr.

[672] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/source_location.hpp
  ```

Public Functions

inline constexpr *std*::uint_least32_t **line**() const noexcept
return the line number represented by this object

inline constexpr char const ***file_name**() const noexcept
return the file name represented by this object

inline constexpr char const ***function_name**() const noexcept
return the name of the function represented by this object, if any

Public Members

char const ***filename**

std::uint_least32_t **line_number**

char const ***functionname**

Public Static Functions

static inline constexpr *std*::uint_least32_t **column**() noexcept
return the column number represented by this object

namespace **assertion**

Typedefs

using **instead** = *hpx*::*source_location*

HPX_ASSERT, HPX_ASSERT_MSG

Defined in header hpx/assert.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_ASSERT(expr)
This macro asserts that *expr* evaluates to true.

If `expr` evaluates to false, The source location and `msg` is being printed along with the expression and additional. Afterwards the program is being aborted. The assertion handler can be customized by calling hpx::assertion::set_assertion_handler().

Asserts are enabled if *HPX_DEBUG* is set. This is the default for `CMAKE_BUILD_TYPE=Debug`

Parameters

- **expr** – The expression to assert on. This can either be an expression that's convertible to bool or a callable which returns bool
- **msg** – The optional message that is used to give further information if the assert fails. This should be convertible to a std::string

HPX_ASSERT_MSG(expr, msg)

See also:

HPX_ASSERT

namespace **hpx**

namespace **assertion**

Typedefs

using **assertion_handler** = void (*)(*hpx*::*source_location* const &loc, const char *expr, *std*::string const &msg)

The signature for an assertion handler.

Functions

void **set_assertion_handler**(*assertion_handler* handler)

Set the assertion handler to be used within a program. If the handler has been set already once, the call to this function will be ignored.

Note: This function is not thread safe

async_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::async

Defined in header hpx/future.hpp[673].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[673] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

Functions

template<typename **F**, typename ...**Ts**>
decltype(auto) **async**(*F* &&f, *Ts*&&... ts)

The function template *async* runs the function *f* asynchronously (potentially in a separate thread which might be a part of a thread pool) and returns an *hpx::future* that will eventually hold the result of that function call. If no policy is defined, *async* behaves as if it is called with policy being hpx::launch::async | *hpx::launch::deferred*. Otherwise, it calls a function *f* with arguments *ts* according to a specific launch policy.

- If the async flag is set (i.e. (policy & *hpx::launch::async*) != 0), then async executes the callable object f on a new thread of execution (with all thread-locals initialized) as if spawned by hpx::thread(std::forward<F>(f), std::forward<Ts>(ts)…), except that if the function f returns a value or throws an exception, it is stored in the shared state accessible through the hpx::future that async returns to the caller.
- If the deferred flag is set (i.e. (policy & *hpx::launch::deferred*) != 0), then async converts f and ts… the same way as by hpx::thread constructor, but does not spawn a new thread of execution. Instead, lazy evaluation is performed: the first call to a non-timed wait function on the hpx::future that async returned to the caller will cause the copy of f to be invoked (as an rvalue) with the copies of ts… (also passed as rvalues) in the current thread (which does not have to be the thread that originally called hpx::async). The result or exception is placed in the shared state associated with the future and only then it is made ready. All further accesses to the same hpx::future will return the result immediately.
- If neither *hpx::launch::async* nor *hpx::launch::deferred*, nor any implementation-defined policy flag is set in policy, the behavior is undefined.

If more than one flag is set, it is implementation-defined which policy is selected. For the default (both the *hpx::launch::async* and *hpx::launch::deferred* flags are set in policy), standard recommends (but doesn't require) utilizing available concurrency, and deferring any additional tasks.

In any case, the call to hpx::async synchronizes-with (as defined in std::memory_order) the call to f, and the completion of f is sequenced-before making the shared state ready. If the async policy is chosen, the associated thread completion synchronizes-with the successful return from the first function that is waiting on the shared state, or with the return of the last function that releases the shared state, whichever comes first. If std::decay<Function>::type or each type in std::decay<Ts>::type is not constructible from its corresponding argument, the program is ill-formed.

Parameters

- **f** – Callable object to call
- **ts** – parameters to pass to f

Returns hpx::future referring to the shared state created by this call to *hpx::async*.

hpx::dataflow

Defined in header hpx/future.hpp[674].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[674] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

Functions

template<typename **F**, typename ...**Ts**>
decltype(auto) **dataflow**(*F* &&f, *Ts*&&... ts)

The function template *dataflow* runs the function f asynchronously (potentially in a separate thread which might be a part of a thread pool) and returns a `hpx::future` that will eventually hold the result of that function call. Its behavior is similar to `hpx::async` with the exception that if one of the arguments is a *future*, then `hpx::dataflow` will wait for the *future* to be ready to launch the thread. Hence, the operation is delayed until all the arguments are ready.

hpx::launch

Defined in header hpx/future.hpp[675].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

struct **launch** : public detail::policy_holder<>

#include <launch_policy.hpp> Launch policies for *hpx::async* etc.

Public Functions

inline constexpr **launch**() noexcept

Default constructor. This creates a launch policy representing all possible launch modes

inline constexpr **launch**(detail::async_policy p) noexcept

Create a launch policy representing asynchronous execution.

inline constexpr **launch**(detail::fork_policy p) noexcept

Create a launch policy representing asynchronous execution. The new thread is executed in a preferred way

inline constexpr **launch**(detail::sync_policy p) noexcept

Create a launch policy representing synchronous execution.

inline constexpr **launch**(detail::deferred_policy p) noexcept

Create a launch policy representing deferred execution.

inline constexpr **launch**(detail::apply_policy p) noexcept

Create a launch policy representing fire and forget execution.

template<typename **F**>
inline constexpr **launch**(detail::select_policy<*F*> const &p) noexcept

Create a launch policy representing fire and forget execution.

template<typename **Launch**, typename **Enable** = *std*::enable_if_t<*hpx*::*traits*::is_launch_policy_v<*Launch*>>>
inline constexpr **launch**(*Launch* l, *threads*::*thread_priority* priority, *threads*::*thread_stacksize* stacksize, *threads*::*thread_schedule_hint* hint) noexcept

[675] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

Public Static Attributes

static const detail::async_policy **async**

Predefined launch policy representing asynchronous execution.

static const detail::fork_policy **fork**

Predefined launch policy representing asynchronous execution.The new thread is executed in a preferred way

static const detail::sync_policy **sync**

Predefined launch policy representing synchronous execution.

static const detail::deferred_policy **deferred**

Predefined launch policy representing deferred execution.

static const detail::apply_policy **apply**

Predefined launch policy representing fire and forget execution.

static const detail::select_policy_generator **select**

Predefined launch policy representing delayed policy selection.

Friends

inline friend *launch* **tag_invoke**(*hpx*::*execution*::*experimental*::with_priority_t, *launch* const &policy, *threads*::*thread_priority* priority) noexcept

inline friend constexpr friend hpx::threads::thread_priority tag_invoke (hpx::execution::exper launch const &policy) noexcept

inline friend *launch* **tag_invoke**(*hpx*::*execution*::*experimental*::with_stacksize_t, *launch* const &policy, *threads*::*thread_stacksize* stacksize) noexcept

inline friend constexpr friend hpx::threads::thread_stacksize tag_invoke (hpx::execution::exper launch const &policy) noexcept

inline friend *launch* **tag_invoke**(*hpx*::*execution*::*experimental*::with_hint_t, *launch* const &policy, *threads*::*thread_schedule_hint* hint) noexcept

inline friend constexpr friend hpx::threads::thread_schedule_hint tag_invoke (hpx::execution::e launch const &policy) noexcept

hpx::post

Defined in header hpx/future.hpp[676].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **F**, typename ...**Ts**>
bool **post**(*F* &&f, *Ts*&&... ts)

Runs the function `f` asynchronously (potentially in a separate thread which might be a part of a thread pool). This is done in a fire-and-forget manner, meaning there is no return value or way to synchronize with the function execution (it does not return an `hpx::future` that would hold the result of that function call).

`hpx::post` is particularly useful when synchronization mechanisms as heavyweight as futures are not desired, and instead, more lightweight mechanisms like latches or atomic variables are preferred. Essentially, the post function enables the launch of a new thread without the overhead of creating a future.

Note: `hpx::post` is similar to `hpx::async` but does not return a future. This is why there is no way of finding out the result/failure of the execution of this function.

hpx::sync

Defined in header hpx/future.hpp[677].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **F**, typename ...**Ts**>
decltype(auto) **sync**(*F* &&f, *Ts*&&... ts)

The function template *sync* runs the function *f* synchronously and returns an *hpx::future* that will eventually hold the result of that function call.

[676] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp
[677] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

async_combinators

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/async_combinators/split_future.hpp

Defined in header hpx/async_combinators/split_future.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename ...**Ts**>
inline *tuple*<*future*<*Ts*>...> **split_future**(*future*<*tuple*<*Ts*...>> &&f)

The function *split_future* is an operator allowing to split a given future of a sequence of values (any tuple, std::pair, or std::array) into an equivalent container of futures where each future represents one of the values from the original future. In some sense this function provides the inverse operation of *when_all*.

Note: The following cases are special:

```
tuple<future<void> > split_future(future<tuple<> > && f);
array<future<void>, 1> split_future(future<array<T, 0> > && f);
```

here the returned futures are directly representing the futures which were passed to the function.

Parameters **f** – [in] A future holding an arbitrary sequence of values stored in a tuple-like container. This facility supports *hpx::tuple<>*, *std::pair<T1, T2>*, and *std::array<T, N>*

Returns Returns an equivalent container (same container type as passed as the argument) of futures, where each future refers to the corresponding value in the input parameter. All of the returned futures become ready once the input future has become ready. If the input future is exceptional, all output futures will be exceptional as well.

template<typename **T**>
inline *std*::vector<*future*<*T*>> **split_future**(*future*<*std*::vector<*T*>> &&f, *std*::size_t size)

The function *split_future* is an operator allowing to split a given future of a sequence of values (any std::vector) into a std::vector of futures where each future represents one of the values from the original std::vector. In some sense this function provides the inverse operation of *when_all*.

Parameters

- **f** – [in] A future holding an arbitrary sequence of values stored in a std::vector.
- **size** – [in] The number of elements the vector will hold once the input future has become ready

Returns Returns a std::vector of futures, where each future refers to the corresponding value in the input parameter. All of the returned futures become ready once the input future has become ready. If the input future is exceptional, all output futures will be exceptional as well.

hpx::wait_all

Defined in header hpx/future.hpp[678].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **InputIter**>
void **wait_all**(*InputIter* first, *InputIter* last)

The function *wait_all* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note: The function *wait_all* returns after all futures have become ready. All input futures are still valid after *wait_all* returns.

Note: The function wait_all will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_all_nothrow* instead.

Parameters

- **first** – The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_all* should wait.
- **last** – The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *wait_all* should wait.

template<typename **R**>
void **wait_all**(*std*::vector<*future*<*R*>> &&futures)

The function *wait_all* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note: The function *wait_all* returns after all futures have become ready. All input futures are still valid after *wait_all* returns.

Note: The function wait_all will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_all_nothrow* instead.

Parameters **futures** – A vector or array holding an arbitrary amount of *future* or *shared_future* objects for which *wait_all* should wait.

template<typename **R**, *std*::size_t **N**>

[678] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

void **wait_all**(*std*::array<*future*<*R*>, *N*> &&futures)

The function *wait_all* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note: The function *wait_all* returns after all futures have become ready. All input futures are still valid after *wait_all* returns.

Note: The function wait_all will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_all_nothrow* instead.

Parameters **futures** – A vector or array holding an arbitrary amount of *future* or *shared_future* objects for which *wait_all* should wait.

template<typename **T**>
void **wait_all**(*hpx*::*future*<*T*> const &f)

The function *wait_all* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note: The function *wait_all* returns after the future has become ready. The input future is still valid after *wait_all* returns.

Note: The function wait_all will rethrow any exceptions captured by the future while becoming ready. If this behavior is undesirable, use *wait_all_nothrow* instead.

Parameters **f** – A *future* or *shared_future* for which *wait_all* should wait.

template<typename ...**T**>
void **wait_all**(*T*&&... futures)

The function *wait_all* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note: The function *wait_all* returns after all futures have become ready. All input futures are still valid after *wait_all* returns.

Note: The function wait_all will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_all_nothrow* instead.

Parameters **futures** – An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_all* should wait.

template<typename **InputIter**>
void **wait_all_n**(*InputIter* begin, *std*::size_t count)

The function *wait_all_n* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note: The function *wait_all_n* returns after all futures have become ready. All input futures are still valid after *wait_all_n* returns.

Note: The function wait_all_n will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_all_n_nothrow* instead.

Parameters

- **`begin`** – The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_all_n* should wait.
- **`count`** – The number of elements in the sequence starting at *first*.

Returns The function *wait_all_n* will return an iterator referring to the first element in the input sequence after the last processed element.

hpx::wait_any

Defined in header hpx/future.hpp[679].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **InputIter**>
void **wait_any**(*InputIter* first, *InputIter* last)

The function *wait_any* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note: The function *wait_any* returns after at least one future has become ready. All input futures are still valid after *wait_any* returns.

Note: The function wait_any will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_any_nothrow* instead.

Parameters

- **`first`** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_any* should wait.
- **`last`** – [in] The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *wait_any* should wait.

[679] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

template<typename **R**>
void **wait_any**(*std*::vector<*future*<*R*>> &futures)

The function *wait_any* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note: The function *wait_any* returns after at least one future has become ready. All input futures are still valid after *wait_any* returns.

Note: The function wait_any will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_any_nothrow* instead.

Parameters **futures** – [in] A vector holding an arbitrary amount of *future* or *shared_future* objects for which *wait_any* should wait.

template<typename **R**, *std*::size_t **N**>
void **wait_any**(*std*::array<*future*<*R*>, *N*> &futures)

The function *wait_any* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note: The function *wait_any* returns after at least one future has become ready. All input futures are still valid after *wait_any* returns.

Note: The function wait_any will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_any_nothrow* instead.

Parameters **futures** – [in] Amn array holding an arbitrary amount of *future* or *shared_future* objects for which *wait_any* should wait.

template<typename ...**T**>
void **wait_any**(*T*&&... futures)

The function *wait_any* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note: The function *wait_any* returns after at least one future has become ready. All input futures are still valid after *wait_any* returns.

Note: The function wait_any will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_any_nothrow* instead.

Parameters **futures** – [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_any* should wait.

template<typename **InputIter**>

void **wait_any_n**(*InputIter* first, *std*::size_t count)

The function *wait_any_n* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note: The function *wait_any_n* returns after at least one future has become ready. All input futures are still valid after *wait_any_n* returns.

Note: The function wait_any_n will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_any_n_nothrow* instead.

Parameters

- **first** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_any_n* should wait.
- **count** – [in] The number of elements in the sequence starting at *first*.

hpx::wait_each

Defined in header hpx/future.hpp[680].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **F**, typename **Future**>
void **wait_each**(*F* &&f, *std*::vector<*Future*> &&futures)

The function *wait_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready. *wait_each* returns after all futures have been become ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **futures** – A vector holding an arbitrary amount of *future* or *shared_future* objects for which *wait_each* should wait.

[680] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

template<typename **F**, typename **Iterator**>
void **wait_each**(*F* &&f, *Iterator* begin, *Iterator* end)

The function *wait_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready. *wait_each* returns after all futures have been become ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **begin** – The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.
- **end** – The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.

template<typename **F**, typename ...**T**>
void **wait_each**(*F* &&f, *T*&&... futures)

The function *wait_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready. *wait_each* returns after all futures have been become ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **futures** – An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_each* should wait.

template<typename **F**, typename **Iterator**>
void **wait_each_n**(*F* &&f, *Iterator* begin, *std*::size_t count)

The function *wait_each* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing. Additionally, the supplied function is called for each of the passed futures as soon as the future has become ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **begin** – The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_each_n* should wait.
- **count** – The number of elements in the sequence starting at *first*.

hpx::wait_some

Defined in header hpx/future.hpp[681].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **InputIter**>
void **wait_some**(*std*::size_t n, *InputIter* first, *InputIter* last)

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The function *wait_some* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note: The function wait_some will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the function to return.
- **first** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- **last** – [in] The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.

template<typename **R**>

[681] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

void **wait_some**(*std*::size_t n, *std*::vector<*future*<*R*>> &&futures)

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The function *wait_some* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note: The function wait_some will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **futures** – [in] A vector holding an arbitrary amount of *future* or *shared_future* objects for which *wait_some* should wait.

template<typename **R**, *std*::size_t **N**>
void **wait_some**(*std*::size_t n, *std*::array<*future*<*R*>, *N*> &&futures)

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The function *wait_some* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note: The function wait_some will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **futures** – [in] An array holding an arbitrary amount of *future* or *shared_future* objects for which *wait_some* should wait.

template<typename ...**T**>
void **wait_some**(*std*::size_t n, *T*&&... futures)

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The function *wait_all* returns after *n* futures have become ready. All input futures are still valid after *wait_some* returns.

Note: The function wait_some will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_nothrow* instead.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **futures** – [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_some* should wait.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

template<typename **InputIter**>
void **wait_some_n**(*std*::size_t n, *InputIter* first, *std*::size_t count)

The function *wait_some_n* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The function *wait_some_n* returns after *n* futures have become ready. All input futures are still valid after *wait_some_n* returns.

Note: The function wait_some_n will rethrow any exceptions captured by the futures while becoming ready. If this behavior is undesirable, use *wait_some_n_nothrow* instead.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **first** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- **count** – [in] The number of elements in the sequence starting at *first*.

hpx::when_all

Defined in header hpx/future.hpp[682].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

[682] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

Functions

template<typename **InputIter**, typename **Container** = vector<*future*<typename *std*::iterator_traits<*InputIter*>::value_type>>>
hpx::*future*<*Container*> **when_all**(*InputIter* first, *InputIter* last)

function *when_all* creates a future object that becomes ready when all elements in a set of *future* and *shared_future* objects become ready. It is an operator allowing to join on the result of all given futures. It AND-composes all given future objects and returns a new future object representing the same list of futures after they finished executing.

Note: Calling this version of *when_all* where first == last, returns a future with an empty container that is immediately ready. Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_all* will not throw an exception, but the futures held in the output collection may.

Parameters

- `first` – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- `last` – [in] The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.

Returns Returns a future holding the same list of futures as has been passed to *when_all*.

- future<Container<future<R>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

template<typename **Range**>
hpx::*future*<*Range*> **when_all**(*Range* &&values)

function *when_all* creates a future object that becomes ready when all elements in a set of *future* and *shared_future* objects become ready. It is an operator allowing to join on the result of all given futures. It AND-composes all given future objects and returns a new future object representing the same list of futures after they finished executing.

Note: Calling this version of *when_all* where the input container is empty, returns a future with an empty container that is immediately ready. Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_all* will not throw an exception, but the futures held in the output collection may.

Parameters `values` – [in] A range holding an arbitrary amount of *future* or *shared_future* objects for which *when_all* should wait.

Returns Returns a future holding the same list of futures as has been passed to when_all.

- future<Container<future<R>>>: If the input cardinality is unknown at compile time and the futures are all of the same type.

template<typename ...**T**>
hpx::*future*<*hpx*::*tuple*<*hpx*::*future*<*T*>...>> **when_all**(*T*&&... futures)

function *when_all* creates a future object that becomes ready when all elements in a set of *future* and *shared_future* objects become ready. It is an operator allowing to join on the result of all given futures. It

AND-composes all given future objects and returns a new future object representing the same list of futures after they finished executing.

Note: Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_all* will not throw an exception, but the futures held in the output collection may.

Parameters **`futures`** – [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *when_all* should wait.

Returns Returns a future holding the same list of futures as has been passed to *when_all*.

- future<tuple<future<T0>, future<T1>, future<T2>…>>: If inputs are fixed in number and are of heterogeneous types. The inputs can be any arbitrary number of future objects.
- future<tuple<>> if *when_all* is called with zero arguments. The returned future will be initially ready.

template<typename **`InputIter`**, typename **`Container`** = vector<*future*<typename *std*::iterator_traits<*InputIter*>::value_type>>>
hpx::*future*<*Container*> **`when_all_n`**(*InputIter* begin, *std*::size_t count)

function *when_all* creates a future object that becomes ready when all elements in a set of *future* and *shared_future* objects become ready. It is an operator allowing to join on the result of all given futures. It AND-composes all given future objects and returns a new future object representing the same list of futures after they finished executing.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: None of the futures in the input sequence are invalidated.

Parameters

- **`begin`** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_all_n* should wait.
- **`count`** – [in] The number of elements in the sequence starting at *first*.

Throws `This` – function will throw errors which are encountered while setting up the requested operation only. Errors encountered while executing the operations delivering the results to be stored in the futures are reported through the futures themselves.

Returns Returns a future holding the same list of futures as has been passed to *when_all_n*.

- future<Container<future<R>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output vector will be the same as given by the input iterator.

hpx::when_any

Defined in header hpx/future.hpp[683].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **InputIter**, typename **Container** = vector<*future*<typename *std*::iterator_traits<*InputIter*>::value_type>>>
future<*when_any_result*<*Container*>> **when_any**(*InputIter* first, *InputIter* last)

function *when_any* creates a future object that becomes when at least one element in a set of *future* and *shared_future* objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.

Parameters

- **first** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_any* should wait.
- **last** – [in] The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *when_any* should wait.

Returns Returns a *when_any_result* holding the same list of futures as has been passed to when_any and an index pointing to a ready future.

- future<*when_any_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

template<typename **Range**>
future<*when_any_result*<*Range*>> **when_any**(*Range* &values)

function *when_any* creates a future object that becomes when at least one element in a set of *future* and *shared_future* objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.

Parameters **values** – [in] A range holding an arbitrary amount of *futures* or *shared_future* objects for which *when_any* should wait.

Returns Returns a *when_any_result* holding the same list of futures as has been passed to when_any and an index pointing to a ready future.

- future<*when_any_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

template<typename ...**T**>
future<*when_any_result*<*tuple*<*future*<*T*>...>>> **when_any**(*T*&&... futures)

function *when_any* creates a future object that becomes when at least one element in a set of *future* and *shared_future* objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given

[683] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.

Parameters **futures** – [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *when_any* should wait.

Returns Returns a *when_any_result* holding the same list of futures as has been passed to when_any and an index pointing to a ready future..

- future<*when_any_result*<tuple<future<T0>, future<T1>…>>>: If inputs are fixed in number and are of heterogeneous types. The inputs can be any arbitrary number of future objects.
- future<*when_any_result*<tuple<>>> if *when_any* is called with zero arguments. The returned future will be initially ready.

template<typename **InputIter**, typename **Container** = vector<*future*<typename *std*::iterator_traits<*InputIter*>::value_type>>>
future<*when_any_result*<*Container*>> **when_any_n**(*InputIter* first, *std*::size_t count)

function *when_any_n* creates a future object that becomes when at least one element in a set of *future* and *shared_future* objects becomes ready. It is a non-deterministic choice operator. It OR-composes all given future objects and returns a new future object representing the same list of futures after one future of that list finishes execution.

Note: None of the futures in the input sequence are invalidated.

Parameters

- **first** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_any_n* should wait.
- **count** – [in] The number of elements in the sequence starting at *first*.

Returns Returns a *when_any_result* holding the same list of futures as has been passed to when_any and an index pointing to a ready future.

- future<*when_any_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

template<typename **Sequence**>

struct **when_any_result**

#include <when_any.hpp> Result type for *when_any*, contains a sequence of futures and an index pointing to a ready future.

Public Members

std::size_t **index**

The index of a future which has become ready.

Sequence **futures**

The sequence of futures as passed to hpx::when_any

hpx::when_each

Defined in header hpx/future.hpp[684].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **F**, typename **Future**>
future<void> **when_each**(*F* &&f, *std*::vector<*Future*> &&futures)

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **futures** – A vector holding an arbitrary amount of *future* or *shared_future* objects for which *wait_each* should wait.

Returns Returns a future representing the event of all input futures being ready.

template<typename **F**, typename **Iterator**>
future<*Iterator*> **when_each**(*F* &&f, *Iterator* begin, *Iterator* end)

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **begin** – The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.

[684] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

- **end** – The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *wait_each* should wait.

Returns Returns a future representing the event of all input futures being ready.

template<typename **F**, typename ...**Ts**>
future<void> **when_each**(*F* &&f, *Ts*&&... futures)

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **futures** – An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_each* should wait.

Returns Returns a future representing the event of all input futures being ready.

template<typename **F**, typename **Iterator**>
future<*Iterator*> **when_each_n**(*F* &&f, *Iterator* begin, *std*::size_t count)

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note: This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

Parameters

- **f** – The function which will be called for each of the input futures once the future has become ready.
- **begin** – The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_each_n* should wait.
- **count** – The number of elements in the sequence starting at *first*.

Returns Returns a future holding the iterator pointing to the first element after the last one.

hpx::when_some

Defined in header hpx/future.hpp[685].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **InputIter**, typename **Container** = vector<*future*<typename *std*::iterator_traits<*InputIter*>::value_type>>>
future<*when_some_result*<*Container*>> **when_some**(*std*::size_t n, Iterator first, Iterator last)

The function *when_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The future returned by the function *when_some* becomes ready when at least *n* argument futures have become ready.

Note: Calling this version of *when_some* where first == last, returns a future with an empty container that is immediately ready. Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some* will not throw an exception, but the futures held in the output collection may.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **first** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- **last** – [in] The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.

Returns Returns a *when_some_result* holding the same list of futures as has been passed to when_some and indices pointing to ready futures.

- future<*when_some_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

template<typename **Range**>
future<*when_some_result*<*Range*>> **when_some**(*std*::size_t n, *Range* &&futures)

The function *when_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

[685] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

Note: The future returned by the function *when_some* becomes ready when at least *n* argument futures have become ready.

Note: Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some* will not throw an exception, but the futures held in the output collection may.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **futures** – [in] A container holding an arbitrary amount of *future* or *shared_future* objects for which *when_some* should wait.

Returns Returns a *when_some_result* holding the same list of futures as has been passed to when_some and indices pointing to ready futures.

- future<*when_some_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

template<typename ...**Ts**>
future<*when_some_result*<*tuple*<*future*<T>...>>> **when_some**(*std*::size_t n, *Ts*&&... futures)

The function *when_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The future returned by the function *when_some* becomes ready when at least *n* argument futures have become ready.

Note: Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some* will not throw an exception, but the futures held in the output collection may.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **futures** – [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *when_some* should wait.

Returns Returns a *when_some_result* holding the same list of futures as has been passed to when_some and an index pointing to a ready future..

- future<*when_some_result*<tuple<future<T0>, future<T1>…>>>: If inputs are fixed in number and are of heterogeneous types. The inputs can be any arbitrary number of future objects.
- future<*when_some_result*<tuple<>>> if *when_some* is called with zero arguments. The returned future will be initially ready.

template<typename **InputIter**, typename **Container** = vector<*future*<typename *std*::iterator_traits<*InputIter*>::value_type>>>
future<*when_some_result*<*Container*>> **when_some_n**(*std*::size_t n, Iterator first, *std*::size_t count)

The function *when_some_n* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note: The future returned by the function *when_some_n* becomes ready when at least *n* argument futures have become ready.

Note: Calling this version of *when_some_n* where count == 0, returns a future with the same elements as the arguments that is immediately ready. Possibly none of the futures in that container are ready. Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some_n* will not throw an exception, but the futures held in the output collection may.

Parameters

- **n** – [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- **first** – [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- **count** – [in] The number of elements in the sequence starting at *first*.

Returns Returns a *when_some_result* holding the same list of futures as has been passed to when_some and indices pointing to ready futures.

- future<*when_some_result*<Container<future<R>>>>: If the input cardinality is unknown at compile time and the futures are all of the same type. The order of the futures in the output container will be the same as given by the input iterator.

template<typename **Sequence**>

struct **when_some_result**

#include <when_some.hpp> Result type for *when_some*, contains a sequence of futures and indices pointing to ready futures.

Public Members

std::vector<*std*::size_t> **indices**

List of indices of futures that have become ready.

Sequence **futures**

The sequence of futures as passed to hpx::when_some.

async_cuda

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/async_cuda/cublas_executor.hpp

Defined in header hpx/async_cuda/cublas_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/async_cuda/cuda_executor.hpp

Defined in header hpx/async_cuda/cuda_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **cuda**

namespace **experimental**

struct **cuda_executor** : public *hpx*::*cuda*::*experimental*::*cuda_executor_base*

Public Functions

inline explicit **cuda_executor**(*std*::size_t device, bool event_mode = true)

inline **~cuda_executor**()

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*post_t*, *cuda_executor* const &exec, *F* &&f, *Ts*&&... ts)

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*async_execute_t*, *cuda_executor* const &exec, *F* &&f, *Ts*&&... ts)

Protected Functions

template<typename **R**, typename ...**Params**, typename ...**Args**>
inline void **post**(*R* (*cuda_function)(*Params*...), *Args*&&... args) const

template<typename **R**, typename ...**Params**, typename ...**Args**>
inline *hpx*::*future*<void> **async**(*R* (*cuda_kernel)(*Params*...), *Args*&&... args) const

struct **cuda_executor_base**

Subclassed by *hpx::cuda::experimental::cuda_executor*

Public Types

using **future_type** = *hpx*::*future*<void>

Public Functions

inline **cuda_executor_base**(*std*::size_t device, bool event_mode)

inline *future_type* **get_future**() const

Protected Attributes

int **device_**

bool **event_mode_**

cudaStream_t **stream_**

std::shared_ptr<*hpx*::*cuda*::*experimental*::target> **target_**

namespace **parallel**

namespace **execution**

async_mpi

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/async_mpi/mpi_executor.hpp

Defined in header hpx/async_mpi/mpi_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **mpi**

namespace **experimental**

struct **executor**

Public Types

using **execution_category** = *hpx*::*execution*::*parallel_execution_tag*

using **executor_parameters_type** = *hpx*::*execution*::*experimental*::*default_parameters*

Public Functions

inline explicit constexpr **executor**(MPI_Comm communicator = MPI_COMM_WORLD)

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*async_execute_t*, *executor* const &exec, *F* &&f, *Ts*&&... ts)

inline *std*::size_t **in_flight_estimate**() const

Private Members

MPI_Comm **communicator_**

namespace **parallel**

namespace **execution**

hpx/async_mpi/transform_mpi.hpp

Defined in header hpx/async_mpi/transform_mpi.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **mpi**

namespace **experimental**

Variables

hpx::*mpi*::*experimental*::*transform_mpi_t* **transform_mpi**

struct **transform_mpi_t** : public *hpx*::*functional*::detail::tag_fallback<*transform_mpi_t*>

Friends

template<typename Sender,
typename F> inline friend constexpr friend auto tag_fallback_invoke (transform_mpi_t,
Sender &&s, F &&f)

template<typename F> inline friend constexpr friend auto tag_fallback_invoke (transform_
F &&f)

async_sycl

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/async_sycl/sycl_executor.hpp

Defined in header hpx/async_sycl/sycl_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **Executor**, typename ...**Ts**>
decltype(auto) **async**(*Executor* &&exec, *hpx*::*sycl*::*experimental*::*sycl_executor*::*queue_function_ptr_t*<*Ts*...> &&f, *Ts*&&... ts)

hpx::async overload for launching sycl queue member functions with an sycl executor

template<typename **Executor**, typename ...**Ts**>
bool **apply**(*Executor* &&exec, *hpx*::*sycl*::*experimental*::*sycl_executor*::*queue_function_ptr_t*<*Ts*...> &&f, *Ts*&&... ts)

hpx::apply overload for launching sycl queue member functions with an sycl executor

namespace **parallel**

namespace **execution**

namespace **sycl**

namespace **experimental**

struct **sycl_executor**

Public Types

using **future_type** = *hpx*::*future*<void>

template<typename ...**Params**>

using **queue_function_ptr_t** = cl::sycl::event (cl::sycl::queue::*)(*std*::conditional_t<*std*::is_trivial_v<*std*::remove_reference_t<*Params*>>, *std*::decay_t<*Params*>, *Params*>...)

Default Implementation without the extra intel code_location parameter. Removes the reference for trivial types to make the function matching easier (see sycl_stream.cpp test)

Public Functions

inline explicit **sycl_executor**(cl::sycl::default_selector selector)

Create a SYCL executor (based on a sycl queue)

~sycl_executor() = default

inline *future_type* **get_future**()

Get future for this command_queue (NOTE will be more efficient if an event is provided — otherwise a dummy kernel must be submitted to get an event)

inline *future_type* **get_future**(cl::sycl::event event)

Get future for that becomes ready when the given event completes.

template<typename ...**Params**>
inline void **post**(*queue_function_ptr_t*<*Params*...> &&queue_member_function, *Params*&&... args)

Invoke queue member function given queue and parameters — do not use event to return a hpx::future (One way)

template<typename ...**Params**>
inline *hpx*::*future*<void> **async_execute**(*queue_function_ptr_t*<*Params*...> &&queue_member_function, *Params*&&... args)

Invoke queue member function given queue and parameters — hpx::future tied to the sycl event / (two way)

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*post_t*, *sycl_executor* &exec, *F* &&f, *Ts*&&... ts)

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*async_execute_t*, *sycl_executor* &exec, *F* &&f, *Ts*&&... ts)

inline cl::sycl::device **get_device**() const

Return the device used by the underlying SYCL queue.

inline cl::sycl::context **get_context**() const

Return the context used by the underlying SYCL queue.

Protected Attributes

cl::sycl::queue **command_queue**

cache

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/cache/local_cache.hpp

Defined in header hpx/cache/local_cache.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

template<typename **Key**, typename **Entry**, typename **UpdatePolicy** = *std*::less<*Entry*>, typename **InsertPolicy** = policies::always<*Entry*>, typename **CacheStorage** = *std*::map<*Key*, *Entry*>, typename **Statistics** = *statistics*::*no_statistics*>
class **local_cache**

#include <hpx/cache/local_cache.hpp> The local_cache implements the basic functionality needed for a local (non-distributed) cache.

Template Parameters

- **Key** – The type of the keys to use to identify the entries stored in the cache
- **Entry** – The type of the items to be held in the cache, must model the CacheEntry concept
- **UpdatePolicy** – A (optional) type specifying a (binary) function object used to sort the cache entries based on their 'age'. The 'oldest' entries (according to this sorting criteria) will be discarded first if the maximum capacity of the cache is reached. The default is std::less<Entry>. The function object will be invoked using 2 entry instances of the type *Entry*. This type must model the UpdatePolicy model.
- **InsertPolicy** – A (optional) type specifying a (unary) function object used to allow global decisions whether a particular entry should be added to the cache or not. The default is *policies::always*, imposing no global insert related criteria on the cache. The function object will be invoked using the entry instance to be inserted into the cache. This type must model the InsertPolicy model.
- **CacheStorage** – A (optional) container type used to store the cache items. The container must be an associative and STL compatible container.The default is a std::map<Key, Entry>.
- **Statistics** – A (optional) type allowing to collect some basic statistics about the operation of the cache instance. The type must conform to the CacheStatistics concept. The default value is the type statistics::no_statistics which does not collect any numbers, but provides empty stubs allowing the code to compile.

Public Types

using **key_type** = *Key*

using **entry_type** = *Entry*

using **update_policy_type** = *UpdatePolicy*

using **insert_policy_type** = *InsertPolicy*

using **storage_type** = *CacheStorage*

using **statistics_type** = *Statistics*

using **value_type** = typename *entry_type*::value_type

using **size_type** = typename *storage_type*::size_type

using **storage_value_type** = typename *storage_type*::value_type

Public Functions

inline explicit **local_cache**(*size_type* max_size = 0, *update_policy_type* const &up = *update_policy_type*(), *insert_policy_type* const &ip = *insert_policy_type*())

Construct an instance of a *local_cache*.

Parameters

- **max_size** – [in] The maximal size this cache is allowed to reach any time. The default is zero (no size limitation). The unit of this value is usually determined by the unit of the values returned by the entry's *get_size* function.
- **up** – [in] An instance of the *UpdatePolicy* to use for this cache. The default is to use a default constructed instance of the type as defined by the *UpdatePolicy* template parameter.
- **ip** – [in] An instance of the *InsertPolicy* to use for this cache. The default is to use a default constructed instance of the type as defined by the *InsertPolicy* template parameter.

local_cache(*local_cache* const &other) = default

local_cache(*local_cache* &&other) = default

local_cache &**operator=**(*local_cache* const &other) = default

local_cache &**operator=**(*local_cache* &&other) = default

~local_cache() = default

inline constexpr *size_type* **size**() const noexcept

Return current size of the cache.

Returns The current size of this cache instance.

inline constexpr *size_type* **capacity**() const noexcept

Access the maximum size the cache is allowed to grow to.

Note: The unit of this value is usually determined by the unit of the return values of the entry's function *entry::get_size*.

Returns The maximum size this cache instance is currently allowed to reach. If this number is zero the cache has no limitation with regard to a maximum size.

inline bool **reserve**(*size_type* max_size)

Change the maximum size this cache can grow to.

Parameters **max_size** – [in] The new maximum size this cache will be allowed to grow to.

Returns This function returns *true* if successful. It returns *false* if the new *max_size* is smaller than the current limit and the cache could not be shrunk to the new maximum size.

inline bool **holds_key**(*key_type* const &k) const

Check whether the cache currently holds an entry identified by the given key.

Note: This function does not call the entry's function *entry::touch*. It just checks if the cache contains an entry corresponding to the given key.

Parameters **k** – [in] The key for the entry which should be looked up in the cache.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline bool **get_entry**(*key_type* const &k, *key_type* &realkey, *entry_type* &val)

Get a specific entry identified by the given key.

Note: The function will call the entry's *entry::touch* function if the value corresponding to the provided key is found in the cache.

Parameters

- **k** – [in] The key for the entry which should be retrieved from the cache.
- **realkey[out]** – Return the full real key found in the cache
- **val** – [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline bool **get_entry**(*key_type* const &k, *entry_type* &val)

Get a specific entry identified by the given key.

Note: The function will call the entry's *entry::touch* function if the value corresponding to the provided key is found in the cache.

Parameters
- **k** – [in] The key for the entry which should be retrieved from the cache.
- **val** – [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline bool **get_entry**(*key_type* const &k, *value_type* &val)

Get a specific entry identified by the given key.

Note: The function will call the entry's *entry::touch* function if the value corresponding to the provided is found in the cache.

Parameters
- **k** – [in] The key for the entry which should be retrieved from the cache
- **val** – [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding value.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline bool **insert**(*key_type* const &k, *value_type* const &val)

Insert a new element into this cache.

Note: This function invokes both, the insert policy as provided to the constructor and the function *entry::insert* of the newly constructed entry instance. If either of these functions returns false the key/value pair doesn't get inserted into the cache and the *insert* function will return *false*. Other reasons for this function to fail (return *false*) are a) the key/value pair is already held in the cache or b) inserting the new value into the cache maxed out its capacity and it was not possible to free any of the existing entries.

Parameters
- **k** – [in] The key for the entry which should be added to the cache.
- **val** – [in] The value which should be added to the cache.

Returns This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

inline bool **insert**(*key_type* const &k, *value_type* &&val)

template<typename **Entry_**, *std*::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Entry_*>, *entry_type*>, int> = 0>
inline bool **insert**(*key_type* const &k, *Entry_* &&e)

Insert a new entry into this cache.

Note: This function invokes both, the insert policy as provided to the constructor and the function *entry::insert* of the provided entry instance. If either of these functions returns false the key/value pair doesn't get inserted into the cache and the *insert* function will return *false*. Other reasons for this function to fail (return *false*) are a) the key/value pair is already held in the cache or b) inserting the new value into the cache maxed out its capacity and it was not possible to free any of the existing entries.

Parameters
- **k** – [in] The key for the entry which should be added to the cache.

- **e** – [in] The entry which should be added to the cache.

Returns This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

template<typename **Value**, *std*::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Value*>, *value_type*>, int> = 0>
inline bool **update**(*key_type* const &k, *Value* &&val)

Update an existing element in this cache.

Note: The function will call the entry's *entry::touch* function if the indexed value is found in the cache.

Note: The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Parameters

- **k** – [in] The key for the value which should be updated in the cache.
- **val** – [in] The value which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.

Returns This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

template<typename **F**, typename **Value**, typename = *std*::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Value*>, *value_type*>>>
inline bool **update_if**(*key_type* const &k, *Value* &&val, *F* &&f)

Update an existing element in this cache.

Note: The function will call the entry's *entry::touch* function if the indexed value is found in the cache.

Note: The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Parameters

- **k** – [in] The key for the value which should be updated in the cache.
- **val** – [in] The value which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.
- **f** – [in] A callable taking two arguments, *k* and the key found in the cache (in that order). If *f* returns true, then the update will continue. If *f* returns false, then the update will not succeed.

Returns This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

template<typename **Entry_**, *std*::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Entry_*>, *entry_type*>, int> = 0>

inline bool **update**(*key_type* const &k, *Entry_* &&e)

Update an existing entry in this cache.

Note: The function will call the entry's *entry::touch* function if the indexed value is found in the cache.

Note: The difference to the other overload of the *insert* function is that this overload replaces the whole cache entry, while the other overload retplaces the cached value only, leaving the cache entry properties untouched.

Parameters
- **k** – [in] The key for the entry which should be updated in the cache.
- **e** – [in] The entry which should be used as a replacement for the existing entry in the cache. Any existing entry is first removed and then this entry is added.

Returns This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

template<typename **Func** = policies::always<*storage_value_type*>>
inline *size_type* **erase**(*Func* &&ep = *Func*())

Remove stored entries from the cache for which the supplied function object returns true.

Parameters **ep** – [in] This parameter has to be a (unary) function object. It is invoked for each of the entries currently held in the cache. An entry is considered for removal from the cache whenever the value returned from this invocation is *true*. Even then the entry might not be removed from the cache as its *entry::remove* function might return false.

Returns This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry::get_size* functions of the removed entries).

inline *size_type* **erase**()

Remove all stored entries from the cache.

Note: All entries are considered for removal, but in the end an entry might not be removed from the cache as its *entry::remove* function might return false. This function is very useful for instance in conjunction with an entry's *entry::remove* function enforcing additional criteria like entry expiration, etc.

Returns This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry::get_size* functions of the removed entries).

inline void **clear**()

Clear the cache.

Unconditionally removes all stored entries from the cache.

inline constexpr *statistics_type* const &**get_statistics**() const noexcept

Allow to access the embedded statistics instance.

Returns This function returns a reference to the statistics instance embedded inside this cache

inline *statistics_type* &**get_statistics**() noexcept

Protected Functions

inline bool **free_space**(long num_free)

Private Types

using **iterator** = typename *storage_type*::iterator

using **const_iterator** = typename *storage_type*::const_iterator

using **heap_type** = *std*::deque<*iterator*>

using **heap_iterator** = typename *heap_type*::iterator

using **adapted_update_policy_type** = *adapt*<*UpdatePolicy*, *iterator*>

using **update_on_exit** = typename *statistics_type*::update_on_exit

Private Members

size_type **max_size_**

size_type **current_size_**

storage_type **store_**

heap_type **entry_heap_**

adapted_update_policy_type **update_policy_**

insert_policy_type **insert_policy_**

statistics_type **statistics_**

template<typename **Func**, typename **Iterator**>

struct **adapt**

Public Functions

inline explicit **adapt**(*Func* const &f)

inline explicit **adapt**(*Func* &&f) noexcept

inline bool **operator()**(*Iterator* const &lhs, *Iterator* const &rhs) const

Public Members

Func **f_**

hpx/cache/lru_cache.hpp

Defined in header hpx/cache/lru_cache.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

template<typename **Key**, typename **Entry**, typename **Statistics** = *statistics*::*no_statistics*>

class **lru_cache**

#include <hpx/cache/lru_cache.hpp> The lru_cache implements the basic functionality needed for a local (non-distributed) LRU cache.

Template Parameters

- **Key** – The type of the keys to use to identify the entries stored in the cache
- **Entry** – The type of the items to be held in the cache.
- **Statistics** – A (optional) type allowing to collect some basic statistics about the operation of the cache instance. The type must conform to the CacheStatistics concept. The default value is the type statistics::no_statistics which does not collect any numbers, but provides empty stubs allowing the code to compile.

Public Types

using **key_type** = *Key*

using **entry_type** = *Entry*

using **statistics_type** = *Statistics*

using **entry_pair** = *std*::pair<*key_type*, *entry_type*>

using **storage_type** = *std*::list<*entry_pair*>

using **map_type** = *std*::map<*Key*, typename *storage_type*::iterator>

using **size_type** = *std*::size_t

Public Functions

inline explicit **lru_cache**(*size_type* max_size = 0)

Construct an instance of a *lru_cache*.

Parameters **max_size** – [in] The maximal size this cache is allowed to reach any time. The default is zero (no size limitation). The unit of this value is usually determined by the unit of the values returned by the entry's *get_size* function.

lru_cache(*lru_cache* const &other) = default

lru_cache(*lru_cache* &&other) = default

lru_cache &**operator=**(*lru_cache* const &other) = default

lru_cache &**operator=**(*lru_cache* &&other) = default

~lru_cache() = default

inline constexpr *size_type* **size**() const noexcept

Return current size of the cache.

Returns The current size of this cache instance.

inline constexpr *size_type* **capacity**() const noexcept

Access the maximum size the cache is allowed to grow to.

Note: The unit of this value is usually determined by the unit of the return values of the entry's function *entry::get_size*.

Returns The maximum size this cache instance is currently allowed to reach. If this number is zero the cache has no limitation with regard to a maximum size.

inline void **reserve**(*size_type* max_size)

Change the maximum size this cache can grow to.

Parameters **max_size** – [in] The new maximum size this cache will be allowed to grow to.

inline bool **holds_key**(*key_type* const &key) const

Check whether the cache currently holds an entry identified by the given key.

Note: This function does not call the entry's function *entry::touch*. It just checks if the cache contains an entry corresponding to the given key.

Parameters **key** – [in] The key for the entry which should be looked up in the cache.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline bool **get_entry**(*key_type* const &key, *key_type* &realkey, *entry_type* &entry)

Get a specific entry identified by the given key.

Note: The function will "touch" the entry and mark it as recently used if the key was found in the cache.

Parameters
- **key** – [in] The key for the entry which should be retrieved from the cache.
- **realkey[out]** – Return the full real key found in the cache
- **entry** – [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

inline bool **get_entry**(*key_type* const &key, *entry_type* const &entry)

Get a specific entry identified by the given key.

Note: The function will "touch" the entry and mark it as recently used if the key was found in the cache.

Parameters
- **key** – [in] The key for the entry which should be retrieved from the cache.
- **entry** – [out] If the entry indexed by the key is found in the cache this value on successful return will be a copy of the corresponding entry.

Returns This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

template<typename **Entry_**, typename = *std*::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Entry_*>, *entry_type*>>>
inline bool **insert**(*key_type* const &key, *Entry_* &&entry)

Insert a new entry into this cache.

Note: This function assumes that the entry is not in the cache already. Inserting an already existing entry is considered undefined behavior

Parameters
- **key** – [in] The key for the entry which should be added to the cache.
- **entry** – [in] The entry which should be added to the cache.

template<typename **Entry_**, typename = *std*::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Entry_*>, *entry_type*>>>
inline void **update**(*key_type* const &key, *Entry_* &&entry)

Update an existing element in this cache.

Note: The function will "touch" the entry and mark it as recently used if the key was found in the cache.

Note: The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Parameters
- **key** – [in] The key for the value which should be updated in the cache.
- **entry** – [in] The entry which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.

template<typename **F**, typename **Entry_**, *std*::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Entry_*>, *entry_type*>, int> = 0>
inline bool **update_if**(*key_type* const &key, *Entry_* &&entry, *F* &&f)

Update an existing element in this cache.

Note: The function will "touch" the entry and mark it as recently used if the key was found in the cache.

Note: The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

Parameters
- **key** – [in] The key for the value which should be updated in the cache.
- **entry** – [in] The value which should be used as a replacement for the existing value in the cache. Any existing cache entry is not changed except for its value.
- **f** – [in] A callable taking two arguments, *k* and the key found in the cache (in that order). If *f* returns true, then the update will continue. If *f* returns false, then the update will not succeed.

Returns This function returns *true* if the entry has been successfully updated, otherwise it returns *false*. If the entry currently is not held by the cache it is added and the return value reflects the outcome of the corresponding insert operation.

template<typename **Func**>
inline *size_type* **erase**(*Func* const &ep)

Remove stored entries from the cache for which the supplied function object returns true.

Parameters **ep** – [in] This parameter has to be a (unary) function object. It is invoked for each of the entries currently held in the cache. An entry is considered for removal from the cache whenever the value returned from this invocation is *true*.

Returns This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry::get_size* functions of the removed entries).

inline *size_type* **erase**()

Remove all stored entries from the cache.

Returns This function returns the overall size of the removed entries (which is the sum of the values returned by the *entry::get_size* functions of the removed entries).

inline *size_type* **clear**()

Clear the cache.

Unconditionally removes all stored entries from the cache.

inline constexpr *statistics_type* const &**get_statistics**() const noexcept

Allow to access the embedded statistics instance.

Returns This function returns a reference to the statistics instance embedded inside this cache inline *statistics_type* &**get_statistics**() noexcept

Private Types

using **update_on_exit** = typename *statistics_type*::update_on_exit

Private Functions

template<typename **Entry_**, typename =
std::enable_if_t<*std*::is_convertible_v<*std*::decay_t<*Entry_*>, *entry_type*>>>
inline void **insert_nonexist**(*key_type* const &key, *Entry_* &&entry)

inline void **touch**(typename *storage_type*::iterator it)

inline void **evict**()

Private Members

size_type **max_size_**

size_type **current_size_** = 0

storage_type **storage_**

map_type **map_**

statistics_type **statistics_**

hpx/cache/entries/entry.hpp

Defined in header hpx/cache/entries/entry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

namespace **entries**

class **entry**

#include <hpx/cache/entries/entry.hpp>

Template Parameters

- **Value** – The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.
- **Derived** – The (optional) type for which this type is used as a base class.

Public Types

using **value_type** = Value

Public Functions

entry() = default

Any cache entry has to be default constructible.

inline explicit **entry**(*value_type* const &val) noexcept(*std*::is_nothrow_copy_constructible_v<*value_type*>)

Construct a new instance of a cache entry holding the given value.

inline explicit **entry**(*value_type* &&val) noexcept

Construct a new instance of a cache entry holding the given value.

inline *value_type* &**get**() noexcept

Get a reference to the stored data value.

Note: This function is part of the CacheEntry concept

inline constexpr *value_type* const &**get**() const noexcept

Public Static Functions

static inline constexpr bool **touch**() noexcept

The function *touch* is called by a cache holding this instance whenever it has been requested (touched).

Note: It is possible to change the entry in a way influencing the sort criteria mandated by the UpdatePolicy. In this case the function should return *true* to indicate this to the cache, forcing to reorder the cache entries.

Note: This function is part of the CacheEntry concept

Returns This function should return true if the cache needs to update it's internal heap. Usually this is needed if the entry has been changed by *touch()* in a way influencing the sort order as mandated by the cache's UpdatePolicy static inline constexpr bool **insert**() noexcept

The function *insert* is called by a cache whenever it is about to be inserted into the cache.

Note: This function is part of the CacheEntry concept

Returns This function should return *true* if the entry should be added to the cache, otherwise it should return *false*.

static inline constexpr bool **remove**() noexcept

The function *remove* is called by a cache holding this instance whenever it is about to be removed from the cache.

Note: This function is part of the CacheEntry concept

Returns The return value can be used to avoid removing this instance from the cache. If the value is *true* it is ok to remove the entry, other wise it will stay in the cache.

static inline constexpr *std*::size_t **get_size**() noexcept

Return the 'size' of this entry. By default the size of each entry is just one (1), which is sensible if the cache has a limit (capacity) measured in number of entries.

Private Members

value_type **value_**

Friends

inline friend bool **operator<**(*entry* const &lhs, *entry* const &rhs) noexcept(noexcept(*std*::declval<*value_type* const&>() < *std*::declval<*value_type* const&>()))

Forwarding operator< allowing to compare entries instead of the values.

hpx/cache/entries/fifo_entry.hpp

Defined in header hpx/cache/entries/fifo_entry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

namespace **entries**

template<typename **Value**>

class **fifo_entry** : public *hpx*::*util*::*cache*::*entries*::*entry*<*Value*, *fifo_entry*<*Value*>>

#include <hpx/cache/entries/fifo_entry.hpp> The fifo_entry type can be used to store arbitrary values in a cache. Using this type as the cache's entry type makes sure that the least recently inserted entries are discarded from the cache first.

Note: The fifo_entry conforms to the CacheEntry concept.

Note: This type can be used to model a 'last in first out' cache policy if it is used with a std::greater as the caches' UpdatePolicy (instead of the default std::less).

Template Parameters **Value** – The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.

Public Functions

fifo_entry() = default

Any cache entry has to be default constructible.

inline explicit **fifo_entry**(*Value* const &val) noexcept(*std*::is_nothrow_constructible_v<*base_type*, *Value* const&>)

Construct a new instance of a cache entry holding the given value.

inline explicit **fifo_entry**(*Value* &&val) noexcept

Construct a new instance of a cache entry holding the given value.

inline constexpr bool **insert**()

The function *insert* is called by a cache whenever it is about to be inserted into the cache.

Note: This function is part of the CacheEntry concept

Returns This function should return *true* if the entry should be added to the cache, otherwise it should return *false*.

inline constexpr *time_point* const &**get_creation_time**() const noexcept

Private Types

using **base_type** = *entry*<*Value*, *fifo_entry*<*Value*>>

using **time_point** = *std*::chrono::steady_clock::time_point

Private Members

time_point **insertion_time_**

Friends

inline friend bool **operator<**(*fifo_entry* const &lhs, *fifo_entry* const &rhs) noexcept(noexcept(*std*::declval<*time_point* const&>() < *std*::declval<*time_point* const&>()))

Compare the 'age' of two entries. An entry is 'older' than another entry if it has been created earlier (FIFO).

hpx/cache/entries/lfu_entry.hpp

Defined in header hpx/cache/entries/lfu_entry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

namespace **entries**

template<typename **Value**>

class **lfu_entry** : public *hpx*::*util*::*cache*::*entries*::*entry*<*Value*, *lfu_entry*<*Value*>>

#include <hpx/cache/entries/lfu_entry.hpp> The lfu_entry type can be used to store arbitrary values in a cache. Using this type as the cache's entry type makes sure that the least frequently used entries are discarded from the cache first.

Note: The lfu_entry conforms to the CacheEntry concept.

Note: This type can be used to model a 'most frequently used' cache policy if it is used with a std::greater as the caches' UpdatePolicy (instead of the default std::less).

Template Parameters **Value** – The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.

Public Functions

lfu_entry() = default

Any cache entry has to be default constructible.

inline explicit **lfu_entry**(*Value* const &val)
noexcept(*std*::is_nothrow_constructible_v<*base_type*, *Value* const&>)

Construct a new instance of a cache entry holding the given value.

inline explicit **lfu_entry**(*Value* &&val) noexcept

Construct a new instance of a cache entry holding the given value.

inline bool **touch**() noexcept

The function *touch* is called by a cache holding this instance whenever it has been requested (touched).

In the case of the LFU entry we store the reference count tracking the number of times this entry has been requested. This which will be used to compare the age of an entry during the invocation of the *operator<()*.

Returns This function should return true if the cache needs to update it's internal heap. Usually this is needed if the entry has been changed by *touch()* in a way influencing the sort order as mandated by the cache's UpdatePolicy

inline constexpr unsigned long const &**get_access_count**() const noexcept

Private Types

using **base_type** = *entry*<*Value*, *lfu_entry*<*Value*>>

Private Members

unsigned long **ref_count_** = 0

Friends

inline friend bool **operator<**(*lfu_entry* const &lhs, *lfu_entry* const &rhs) noexcept

Compare the 'age' of two entries. An entry is 'older' than another entry if it has been accessed less frequently (LFU).

hpx/cache/entries/lru_entry.hpp

Defined in header hpx/cache/entries/lru_entry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

namespace **entries**

template<typename **Value**>

class **lru_entry** : public *hpx*::*util*::*cache*::*entries*::*entry*<*Value*, *lru_entry*<*Value*>>

#include <hpx/cache/entries/lru_entry.hpp> The lru_entry type can be used to store arbitrary values in a cache. Using this type as the cache's entry type makes sure that the least recently used entries are discarded from the cache first.

Note: The lru_entry conforms to the CacheEntry concept.

Note: This type can be used to model a 'most recently used' cache policy if it is used with a std::greater as the caches' UpdatePolicy (instead of the default std::less).

Template Parameters Value – The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.

Public Functions

inline **lru_entry**()

Any cache entry has to be default constructible.

inline explicit **lru_entry**(*Value* const &val) noexcept(*std*::is_nothrow_constructible_v<*base_type*, *Value* const&>)

Construct a new instance of a cache entry holding the given value.

inline explicit **lru_entry**(*Value* &&val) noexcept

Construct a new instance of a cache entry holding the given value.

inline bool **touch**()

The function *touch* is called by a cache holding this instance whenever it has been requested (touched).

In the case of the LRU entry we store the time of the last access which will be used to compare the age of an entry during the invocation of the *operator<()*.

Returns This function should return true if the cache needs to update it's internal heap. Usually this is needed if the entry has been changed by *touch()* in a way influencing the sort order as mandated by the cache's UpdatePolicy

inline constexpr *time_point* const &**get_access_time**() const noexcept

Returns the last access time of the entry.

Private Types

using **base_type** = *entry*<*Value*, *lru_entry*<*Value*>>

using **time_point** = *std*::chrono::steady_clock::time_point

Private Members

time_point **access_time_**

Friends

inline friend bool **operator<**(*lru_entry* const &lhs, *lru_entry* const &rhs) noexcept(noexcept(*std*::declval<*time_point* const&>() < *std*::declval<*time_point* const&>()))

Compare the 'age' of two entries. An entry is 'older' than another entry if it has been accessed less recently (LRU).

hpx/cache/entries/size_entry.hpp

Defined in header hpx/cache/entries/size_entry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

namespace **entries**

class **size_entry**

#include <hpx/cache/entries/size_entry.hpp> The size_entry type can be used to store values in a cache which have a size associated (such as files, etc.). Using this type as the cache's entry type makes sure that the entries with the biggest size are discarded from the cache first.

Note: The size_entry conforms to the CacheEntry concept.

Note: This type can be used to model a 'discard smallest first' cache policy if it is used with a std::greater as the caches' UpdatePolicy (instead of the default std::less).

Template Parameters

- **Value** – The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.
- **Derived** – The (optional) type for which this type is used as a base class.

Public Functions

size_entry() = default

Any cache entry has to be default constructible.

inline explicit **size_entry**(Value const &val, *std*::size_t size = 0) noexcept(*std*::is_nothrow_constructible_v<*base_type*, Value const&>)

Construct a new instance of a cache entry holding the given value.

inline explicit **size_entry**(Value &&val, *std*::size_t size = 0) noexcept

Construct a new instance of a cache entry holding the given value.

inline constexpr *std*::size_t **get_size**() const noexcept

Return the 'size' of this entry.

Private Types

using **derived_type** = typename detail::size_derived<Value, Derived>::type

using **base_type** = *entry*<Value, *derived_type*>

Private Members

std::size_t **size_** = 0

Friends

inline friend constexpr friend bool operator< (size_entry const &lhs, size_entry const &rhs) noexcept

Compare the 'age' of two entries. An entry is 'older' than another entry if it has a bigger size.

hpx/cache/statistics/local_statistics.hpp

Defined in header hpx/cache/statistics/local_statistics.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

namespace **cache**

namespace **statistics**

class **local_statistics** : public *hpx*::*util*::*cache*::*statistics*::*no_statistics*

Public Functions

local_statistics() = default

inline constexpr *std*::size_t **hits**() const noexcept

inline constexpr *std*::size_t **misses**() const noexcept

inline constexpr *std*::size_t **insertions**() const noexcept

inline constexpr *std*::size_t **evictions**() const noexcept

inline *std*::size_t **hits**(bool reset) noexcept

inline *std*::size_t **misses**(bool reset) noexcept

inline *std*::size_t **insertions**(bool reset) noexcept

inline *std*::size_t **evictions**(bool reset) noexcept

inline void **got_hit**() noexcept

The function *got_hit* will be called by a cache instance whenever a entry got touched.

inline void **got_miss**() noexcept

The function *got_miss* will be called by a cache instance whenever a requested entry has not been found in the cache.

inline void **got_insertion**() noexcept

The function *got_insertion* will be called by a cache instance whenever a new entry has been inserted.

inline void **got_eviction**() noexcept

The function *got_eviction* will be called by a cache instance whenever an entry has been removed from the cache because a new inserted entry let the cache grow beyond its capacity.

inline void **clear**() noexcept

Reset all statistics.

Private Members

std::size_t **hits_** = 0

std::size_t **misses_** = 0

std::size_t **insertions_** = 0

std::size_t **evictions_** = 0

Private Static Functions

static inline *std*::size_t **get_and_reset**(*std*::size_t &value, bool reset) noexcept

hpx/cache/statistics/no_statistics.hpp

Defined in header hpx/cache/statistics/no_statistics.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_CACHE_METHOD_UNSCOPED_ENUM_DEPRECATION_MSG

namespace **hpx**

namespace **util**

namespace **cache**

namespace **statistics**

Enums

enum class **method**

Values:

enumerator **get_entry**

enumerator **insert_entry**

enumerator **update_entry**

enumerator **erase_entry**

Variables

constexpr *method* **method_get_entry** = *method*::*get_entry*

constexpr *method* **method_insert_entry** = *method*::*insert_entry*

constexpr *method* **method_update_entry** = *method*::*update_entry*

constexpr *method* **method_erase_entry** = *method*::*erase_entry*

class **no_statistics**

Subclassed by *hpx::util::cache::statistics::local_statistics*

Public Static Functions

static inline constexpr void **got_hit**() noexcept

The function *got_hit* will be called by a cache instance whenever a entry got touched.

static inline constexpr void **got_miss**() noexcept

The function *got_miss* will be called by a cache instance whenever a requested entry has not been found in the cache.

static inline constexpr void **got_insertion**() noexcept

The function *got_insertion* will be called by a cache instance whenever a new entry has been inserted.

static inline constexpr void **got_eviction**() noexcept

The function *got_eviction* will be called by a cache instance whenever an entry has been removed from the cache because a new inserted entry let the cache grow beyond its capacity.

static inline constexpr void **clear**() noexcept

Reset all statistics.

static inline constexpr *std*::int64_t **get_get_entry_count**(bool) noexcept

The function *get_get_entry_count* returns the number of invocations of the get_entry() API function of the cache.

static inline constexpr *std*::int64_t **get_insert_entry_count**(bool) noexcept

The function *get_insert_entry_count* returns the number of invocations of the insert_entry() API function of the cache.

static inline constexpr *std*::int64_t **get_update_entry_count**(bool) noexcept

The function *get_update_entry_count* returns the number of invocations of the update_entry() API function of the cache.

static inline constexpr *std*::int64_t **get_erase_entry_count**(bool) noexcept

The function *get_erase_entry_count* returns the number of invocations of the erase() API function of the cache.

static inline constexpr *std*::int64_t **get_get_entry_time**(bool) noexcept

The function *get_get_entry_time* returns the overall time spent executing of the get_entry() API function of the cache.

static inline constexpr *std*::int64_t **get_insert_entry_time**(bool) noexcept

The function *get_insert_entry_time* returns the overall time spent executing of the insert_entry() API function of the cache.

static inline constexpr *std*::int64_t **get_update_entry_time**(bool) noexcept

The function *get_update_entry_time* returns the overall time spent executing of the update_entry() API function of the cache.

static inline constexpr *std*::int64_t **get_erase_entry_time**(bool) noexcept

The function *get_erase_entry_time* returns the overall time spent executing of the erase() API function of the cache.

struct **update_on_exit**

#include <no_statistics.hpp> Helper class to update timings and counts on function exit.

Public Functions

inline constexpr **update_on_exit**(*no_statistics* const&, *method*) noexcept

compute_local

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/compute_local/vector.hpp

Defined in header hpx/compute_local/vector.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **compute**

Functions

template<typename **T**, typename **Allocator**>
void **swap**(*vector*<*T*, *Allocator*> &x, *vector*<*T*, *Allocator*> &y) noexcept

Effects: x.swap(y);.

template<typename **T**, typename **Allocator** = *std*::allocator<*T*>>

class **vector**

Public Types

using **value_type** = *T*

Member types (FIXME: add reference to std.

using **allocator_type** = *Allocator*

using **access_target** = typename *alloc_traits*::access_target

using **size_type** = *std*::size_t

using **difference_type** = *std*::ptrdiff_t

using **reference** = typename *alloc_traits*::reference

using **const_reference** = typename *alloc_traits*::const_reference

using **pointer** = typename *alloc_traits*::pointer

using **const_pointer** = typename *alloc_traits*::const_pointer

using **iterator** = detail::iterator<*T*, *Allocator*>

using **const_iterator** = detail::iterator<*T* const, *Allocator*>

using **reverse_iterator** = detail::reverse_iterator<*T*, *Allocator*>

using **const_reverse_iterator** = detail::const_reverse_iterator<*T*, *Allocator*>

Public Functions

inline explicit **vector**(*Allocator* const &alloc = *Allocator*())

inline **vector**(*size_type* count, *T* const &value, *Allocator* const &alloc = *Allocator*())

inline explicit **vector**(*size_type* count, *Allocator* const &alloc = *Allocator*())

template<typename **InIter**, typename **Enable** = typename *std*::enable_if<*hpx*::*traits*::is_input_iterator<*InIter*>::value>::type>
inline **vector**(*InIter* first, *InIter* last, *Allocator* const &alloc)

inline **vector**(*vector* const &other)

inline **vector**(*vector* const &other, *Allocator* const &alloc)

inline **vector**(*vector* &&other) noexcept inline **vector**(*vector* &&other, *Allocator* const &alloc)

inline **vector**(*std*::initializer_list<*T*> init, *Allocator* const &alloc)

inline **~vector**()

inline *vector* &**operator=**(*vector* const &other)

inline *vector* &**operator=**(*vector* &&other) noexcept

inline *allocator_type* **get_allocator**() const noexcept

Returns the allocator associated with the container.

inline *reference* **operator[]**(*size_type* pos)

inline *const_reference* **operator[]**(*size_type* pos) const

inline *pointer* **data**() noexcept

Returns pointer to the underlying array serving as element storage. The pointer is such that range [*data()*; *data()* + size()) is always a valid range, even if the container is empty (*data()* is not dereference-able in that case).

inline *const_pointer* **data**() const noexcept

Returns pointer to the underlying array serving as element storage. The pointer is such that range [*data()*; *data()* + size()) is always a valid range, even if the container is empty (*data()* is not dereference-able in that case).

inline *T* ***device_data**() const noexcept

Returns a raw pointer corresponding to the address of the data allocated on the device.

inline *std*::size_t **size**() const noexcept

inline *std*::size_t **capacity**() const noexcept

inline bool **empty**() const noexcept

Returns: size() == 0.

inline void **resize**(*size_type*, *T* const&)

Effects: If size <= size(), equivalent to calling pop_back() size() - size times. If size() < size, appends size - size() copies of val to the sequence.

Requires: T shall be CopyInsertable into *this.

Remarks: If an exception is thrown there are no effects.

inline *iterator* **begin**() noexcept

inline *iterator* **end**() noexcept

inline *const_iterator* **cbegin**() const noexcept

inline *const_iterator* **cend**() const noexcept

inline *const_iterator* **begin**() const noexcept

inline *const_iterator* **end**() const noexcept

inline void **swap**(*vector* &other) noexcept

Effects: Exchanges the contents and capacity() of *this with that of x.

Complexity: Constant time.

inline void **clear**() noexcept

Effects: Erases all elements in the range [begin(),end()). Destroys all elements in 'a'. Invalidates all references, pointers, and iterators referring to the elements of a and may invalidate the past-the-end iterator.

Post: a.empty() returns true.

Complexity: Linear.

Public Static Functions

static inline void **resize**(*size_type*)

Effects: If size <= size(), equivalent to calling pop_back() size()
- size times. If size() < size, appends size - size() default-inserted elements to the sequence.
Requires: T shall be MoveInsertable and DefaultInsertable into *this.

Remarks: If an exception is thrown other than by the move constructor of a non-CopyInsertable T there are no effects.

Private Types

using **alloc_traits** = *traits*::allocator_traits<*Allocator*>

Private Members

size_type **size_**

size_type **capacity_**

allocator_type **alloc_**

pointer **data_**

hpx/compute_local/host/block_executor.hpp

Defined in header hpx/compute_local/host/block_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **Executor**>

struct
`hpx::parallel::execution::`**executor_execution_category**<*compute*::*host*::*block_executor*<*Executor*>>

Public Types

using **type** = *hpx*::*execution*::*parallel_execution_tag*

template<typename **Executor**>

struct **is_one_way_executor**<*compute*::*host*::*block_executor*<*Executor*>> : public true_type

template<typename **Executor**>

struct **is_two_way_executor**<*compute*::*host*::*block_executor*<*Executor*>> : public true_type

template<typename **Executor**>

struct **is_bulk_one_way_executor**<*compute*::*host*::*block_executor*<*Executor*>> : public true_type

template<typename **Executor**>

struct **is_bulk_two_way_executor**<*compute*::*host*::*block_executor*<*Executor*>> : public true_type

namespace **hpx**

namespace **compute**

namespace **host**

template<typename **Executor** = *hpx*::*parallel*::*execution*::*restricted_thread_pool_executor*>

struct **block_executor**

#include <block_executor.hpp> The block executor can be used to build NUMA aware programs. It will distribute work evenly across the passed targets

Template Parameters **Executor** – The underlying executor to use

Public Types

using **executor_parameters_type** = *hpx*::*execution*::*experimental*::*default_parameters*

Public Functions

inline explicit **block_executor**(*std*::vector<*host*::target> const &targets, *threads*::*thread_priority* priority = *threads*::*thread_priority*::*high*, *threads*::*thread_stacksize* stacksize = *threads*::*thread_stacksize*::*default_*, *threads*::*thread_schedule_hint* schedulehint = {})

inline explicit **block_executor**(*std*::vector<*host*::target> &&targets)

inline **block_executor**(*block_executor* const &other)

inline **block_executor**(*block_executor* &&other) noexcept inline *block_executor* &**operator=**(*block_executor* const &other)

inline *block_executor* &**operator=**(*block_executor* &&other) noexcept

inline *std*::vector<*host*::target> const &**targets**() const noexcept

Private Functions

inline auto **get_next_executor**() const

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*post_t*, *block_executor* const &exec, *F* &&f, *Ts*&&... ts)

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*async_execute_t*, *block_executor* const &exec, *F* &&f, *Ts*&&... ts)

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*sync_execute_t*, *block_executor* const &exec, *F* &&f, *Ts*&&... ts)

template<typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) **bulk_async_execute_impl**(*F* &&f, *Shape* const &shape, *Ts*&&... ts) const

template<typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_async_execute_t*, *block_executor* const &exec, *F* &&f, *Shape* const &shape, *Ts*&&... ts)

template<typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) **bulk_sync_execute_impl**(*F* &&f, *Shape* const &shape, *Ts*&&... ts) const

template<typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_sync_execute_t*, *block_executor* const &exec, *F* &&f, *Shape* const &shape, *Ts*&&... ts)

inline void **init_executors**()

Private Members

std::vector<*host*::target> **targets_**

mutable *std*::atomic<*std*::size_t> **current_**

std::vector<*Executor*> **executors_**

threads::*thread_priority* **priority_** = *threads*::*thread_priority*::*high* *threads*::*thread_stacksize* **stacksize_** = *threads*::*thread_stacksize*::*default_*

threads::*thread_schedule_hint* **schedulehint_** = {}

namespace **parallel**

namespace **execution**

template<typename Executor> block_executor< Executor > >

Public Types

using **type** = *hpx*::*execution*::*parallel_execution_tag*

template<typename Executor> block_executor< Executor > > : public true_type

template<typename Executor> block_executor< Executor > > : public true_type

template<typename Executor> block_executor< Executor > > : public true_type

template<typename Executor> block_executor< Executor > > : public true_type

hpx/compute_local/host/block_fork_join_executor.hpp

Defined in header hpx/compute_local/host/block_fork_join_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

class **block_fork_join_executor**

#include <block_fork_join_executor.hpp> An executor with fork-join (blocking) semantics.

The *block_fork_join_executor* creates on construction a set of worker threads that are kept alive for the duration of the executor. Copying the executor has reference semantics, i.e. copies of a fork_join_executor hold a reference to the worker threads of the original instance. Scheduling work through the executor concurrently from different threads is undefined behaviour.

The executor keeps a set of worker threads alive for the lifetime of the executor, meaning other work will not be executed while the executor is busy or waiting for work. The executor has a customizable delay after which it will yield to other work. Since starting and resuming the worker threads is a slow operation the executor should be reused whenever possible for multiple adjacent parallel algorithms or invocations of bulk_(a)sync_execute.

This behaviour is similar to the plain *fork_join_executor* except that the *block_fork_join_executor* creates a hierarchy of fork_join_executors, one for each target used to initialize it.

Public Functions

inline explicit **block_fork_join_executor**(*threads*::*thread_priority* priority = *threads*::*thread_priority*::*bound*, *threads*::*thread_stacksize* stacksize = *threads*::*thread_stacksize*::*small_*, fork_join_executor::loop_schedule const schedule = fork_join_executor::loop_schedule::static_, *std*::chrono::nanoseconds yield_delay = *std*::chrono::milliseconds(1))

Construct a *block_fork_join_executor*.

Note: This constructor will create one fork_join_executor for each numa domain

Parameters

- **priority** – The priority of the worker threads.
- **stacksize** – The stacksize of the worker threads. Must not be nostack.
- **schedule** – The loop schedule of the parallel regions.
- **yield_delay** – The time after which the executor yields to other work if it has not received any new work for execution.

inline explicit **block_fork_join_executor**(*std*::vector<*compute*::*host*::target> const &targets, *threads*::*thread_priority* priority = *threads*::*thread_priority*::*bound*, *threads*::*thread_stacksize* stacksize = *threads*::*thread_stacksize*::*small_*, fork_join_executor::loop_schedule const schedule = fork_join_executor::loop_schedule::static_, *std*::chrono::nanoseconds yield_delay = *std*::chrono::milliseconds(1))

Construct a *block_fork_join_executor*.

Note: This constructor will create one fork_join_executor for each given target

Parameters

- **targets** – The list of targets to use for thread placement
- **priority** – The priority of the worker threads.
- **stacksize** – The stacksize of the worker threads. Must not be nostack.
- **schedule** – The loop schedule of the parallel regions.
- **yield_delay** – The time after which the executor yields to other work if it has not received any new work for execution.

template<typename **F**, typename **S**, typename ...**Ts**>

inline void **bulk_sync_execute_helper**(*F* &&f, *S* const &shape, *Ts*&&... ts)

template<typename **F**, typename **S**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_async_execute_t*, *block_fork_join_executor* &exec, *F* &&f, *S* const &shape, *Ts*&&... ts)

template<typename ...**Fs**>
inline void **sync_invoke_helper**(*Fs*&&... fs) const

template<typename **F**, typename ...**Fs**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*sync_invoke_t*, *block_fork_join_executor* const &exec, *F* &&f, *Fs*&&... fs)

template<typename **F**, typename ...**Fs**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*async_invoke_t*, *block_fork_join_executor* const &exec, *F* &&f, *Fs*&&... fs)

template<typename **Tag**>
inline decltype(auto) friend **tag_invoke**(*Tag* tag, *block_fork_join_executor* const &exec) noexcept

Private Members

fork_join_executor **exec_**

std::vector<fork_join_executor> **block_execs_**

Private Static Functions

static inline *hpx*::*threads*::mask_type **cores_for_targets**(*std*::vector<*compute*::*host*::target> const &targets)

Friends

template<typename **F**, typename **S**, typename ...**Ts**>
inline friend void **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_sync_execute_t*, *block_fork_join_executor* &exec, *F* &&f, *S* const &shape, *Ts*&&... ts)

template<typename **Tag**, typename **Property**>
inline friend *block_fork_join_executor* **tag_invoke**(*Tag* tag, *block_fork_join_executor* const &exec, *Property* &&prop) noexcept

namespace **parallel**

namespace **execution**

config

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/config/endian.hpp

Defined in header hpx/config/endian.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

coroutines

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/coroutines/thread_enums.hpp

Defined in header hpx/coroutines/thread_enums.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Enums

enum class **thread_schedule_state** : *std*::int8_t

The *thread_schedule_state* enumerator encodes the current state of a *thread* instance

Values:

enumerator **unknown**

enumerator **active**

thread is currently active (running, has resources)

enumerator **pending**

thread is pending (ready to run, but no hardware resource available)

enumerator **suspended**

thread has been suspended (waiting for synchronization event, but still known and under control of the thread-manager)

enumerator **depleted**

thread has been depleted (deeply suspended, it is not known to the thread-manager) enumerator **terminated**

thread has been stopped an may be garbage collected

enumerator **staged**

this is not a real thread state, but allows to reference staged task descriptions, which eventually will be converted into thread objects

enumerator **pending_do_not_schedule**

this is not a real thread state, but allows to create a thread in pending state without scheduling it (internal, do not use)

enumerator **pending_boost**

this is not a real thread state, but allows to suspend a thread in pending state without high priority rescheduling

enumerator **deleted**

thread has been stopped and was deleted

enum class **thread_priority** : *std*::int8_t

This enumeration lists all possible thread-priorities for HPX threads.

Values:

enumerator **unknown**

enumerator **default_**

Will assign the priority of the task to the default (normal) priority.

enumerator **low**

Task goes onto a special low priority queue and will not be executed until all high/normal priority tasks are done, even if they are added after the low priority task.

enumerator **normal**

Task will be executed when it is taken from the normal priority queue, this is usually a first in-first-out ordering of tasks (depending on scheduler choice). This is the default priority.

enumerator **high_recursive**

The task is a high priority task and any child tasks spawned by this task will be made high priority as well - unless they are specifically flagged as non default priority.

enumerator **boost**

Same as *thread_priority_high* except that the thread will fall back to *thread_priority_normal* if resumed after being suspended.

enumerator **high**

Task goes onto a special high priority queue and will be executed before normal/low priority tasks are taken (some schedulers modify the behavior slightly and the documentation for those should be consulted).

enumerator **bound**

Task goes onto a special high priority queue and will never be stolen by another thread after initial assignment. This should be used for thread placement tasks such as OpenMP type for loops.

enum class **thread_restart_state** : *std*::int8_t

The *thread_restart_state* enumerator encodes the reason why a thread is being restarted

Values:

enumerator **unknown**

enumerator **signaled**

The thread has been signaled.

enumerator **timeout**

The thread has been reactivated after a timeout.

enumerator **terminate**

The thread needs to be terminated.

enumerator **abort**

The thread needs to be aborted.

enum class **thread_stacksize** : *std*::int8_t

A *thread_stacksize* references any of the possible stack-sizes for HPX threads.

Values:

enumerator **unknown**

enumerator **small_**

use small stack size (the underscore is to work around 'small' being defined to char on Windows)

enumerator **medium**

use medium sized stack size

enumerator **large**

use large stack size

enumerator **huge**

use very large stack size

enumerator **nostack**

this thread does not suspend (does not need a stack)

enumerator **current**

use size of current thread's stack

enumerator **default_**

use default stack size

enumerator **minimal**

use minimally stack size

enumerator **maximal**

use maximally stack size

enum class **thread_schedule_hint_mode** : *std*::int8_t

The type of hint given when creating new tasks.

Values:

enumerator **none**

A hint that leaves the choice of scheduling entirely up to the scheduler.

enumerator **thread**

A hint that tells the scheduler to prefer scheduling a task on the local thread number associated with this hint. Local thread numbers are indexed from zero. It is up to the scheduler to decide how to interpret thread numbers that are larger than the number of threads available to the scheduler. Typically thread numbers will wrap around when too large.

enumerator **numa**

A hint that tells the scheduler to prefer scheduling a task on the NUMA domain associated with this hint. NUMA domains are indexed from zero. It is up to the scheduler to decide how to interpret NUMA domain indices that are larger than the number of available NUMA domains to the scheduler. Typically indices will wrap around when too large.

enum class **thread_placement_hint** : *std*::int8_t

The type of hint given to the scheduler related to thread placement

The type of hint given to the scheduler related running a thread as a child directly in the context of the parent thread

Values:

enumerator **none**

No hint is specified. The implementation is free to chose what placement methods to use.

enumerator **depth_first**

A hint that tells the scheduler to prefer spreading thread placement on a depth-first basis (i.e. consecutively scheduled threads are placed on the same core).

enumerator **breadth_first**

A hint that tells the scheduler to prefer spreading thread placement on a breadth-first basis (i.e. consecutively scheduled threads are placed on the neighboring cores).

enumerator **depth_first_reverse**

A hint that tells the scheduler to prefer spreading thread placement on a depth-first basis (i.e. consecutively scheduled threads are placed on the same core). Threads are being scheduled in reverse order.

enumerator **breadth_first_reverse**

A hint that tells the scheduler to prefer spreading thread placement on a breadth-first basis (i.e. consecutively scheduled threads are placed on the neighboring cores). Threads are being scheduled in reverse order.

enum class **thread_sharing_hint** : *std*::int8_t

The type of hint given to the scheduler related to whether it is ok to share the invoked function object between threads

Values:

enumerator **none**

No hint is specified. The implementation is free to chose what sharing methods to use.

enumerator **do_not_share_function**

A hint that tells the scheduler to avoid sharing the given function (object) between threads.

enumerator **do_not_combine_tasks**

A hint that tells the scheduler to avoid combining tasks on the same thread. This is important for tasks that may synchronize between each other, which could lead to deadlocks if those tasks are combined running by the same thread.

enum class **thread_execution_hint** : *std*::int8_t

Values:

enumerator **none**

No hint is specified. Always run the thread in its own execution environment.

enumerator **run_as_child**

Attempt to run the thread in the execution context of the parent thread.

Functions

std::ostream &**operator<<**(*std*::ostream &os, *thread_schedule_state* t)

char const ***get_thread_state_name**(*thread_schedule_state* state) noexcept

Returns the name of the given state.

Get the readable string representing the name of the given thread_state constant.

Parameters **state** – this represents the thread state.

std::ostream &**operator<<**(*std*::ostream &os, *thread_priority* t) char const ***get_thread_priority_name**(*thread_priority* priority) noexcept

Return the thread priority name.

Get the readable string representing the name of the given thread_priority constant.

Parameters **priority** – this represents the thread priority.

std::ostream &**operator<<**(*std*::ostream &os, *thread_restart_state* t)

char const ***get_thread_state_ex_name**(*thread_restart_state* state) noexcept

Get the readable string representing the name of the given thread_restart_state constant.

char const ***get_thread_state_name**(thread_state state) noexcept

Get the readable string representing the name of the given thread_state constant.

std::ostream &**operator<<**(*std*::ostream &os, *thread_stacksize* t)

char const ***get_stack_size_enum_name**(*thread_stacksize* size) noexcept

Returns the stack size name.

Get the readable string representing the given stack size constant.

Parameters **size** – this represents the stack size

constexpr bool **do_not_share_function**(*thread_sharing_hint* hint) noexcept

constexpr bool **do_not_combine_tasks**(*thread_sharing_hint* hint) noexcept

constexpr *thread_sharing_hint* **operator|**(*thread_sharing_hint* lhs, *thread_sharing_hint* rhs) noexcept

constexpr bool **run_as_child**(*thread_execution_hint* hint) noexcept

Variables

constexpr *thread_execution_hint* **default_runs_as_child_hint** = *thread_execution_hint*::*none*

Default value to use for runs-as-child mode (if *true*, then futures will attempt to execute associated threads directly if they have not started running).

struct **thread_schedule_hint**

#include <thread_enums.hpp> A hint given to a scheduler to guide where a task should be scheduled.

A scheduler is free to ignore the hint, or modify the hint to suit the resources available to the scheduler.

Public Functions

inline constexpr **thread_schedule_hint**() noexcept

Construct a default hint with mode thread_schedule_hint_mode::none.

inline explicit constexpr **thread_schedule_hint**(*std*::int16_t thread_hint, *thread_placement_hint* placement = *thread_placement_hint*::*none*, *thread_execution_hint* runs_as_child = *default_runs_as_child_hint*, *thread_sharing_hint* sharing = *thread_sharing_hint*::*none*) noexcept

Construct a hint with mode thread_schedule_hint_mode::thread and the given hint as the local thread number.

inline constexpr **thread_schedule_hint**(*thread_schedule_hint_mode* mode, *std*::int16_t hint, *thread_placement_hint* placement = *thread_placement_hint*::*none*, *thread_execution_hint* runs_as_child = *default_runs_as_child_hint*, *thread_sharing_hint* sharing = *thread_sharing_hint*::*none*) noexcept

Construct a hint with the given mode and hint. The numerical hint is unused when the mode is thread_schedule_hint_mode::none.

inline constexpr *thread_placement_hint* **placement_mode**() const noexcept

inline void **placement_mode**(*thread_placement_hint* bits) noexcept

inline constexpr *thread_sharing_hint* **sharing_mode**() const noexcept

inline void **sharing_mode**(*thread_sharing_hint* bits) noexcept

inline constexpr *thread_execution_hint* **runs_as_child_mode**() const noexcept

inline void **runs_as_child_mode**(*thread_execution_hint* bits) noexcept

Public Members

std::int16_t **hint** = -1

The hint associated with the mode. The interpretation of this hint depends on the given mode.

thread_schedule_hint_mode **mode** = *thread_schedule_hint_mode*::*none*

The mode of the scheduling hint.

std::uint8_t **placement_mode_bits**

The mode of the desired thread placement.

std::uint8_t **sharing_mode_bits**

The mode of the desired sharing hint.

std::uint8_t **runs_as_child_mode_bits**

The thread will run as a child directly in the context of the current thread

hpx/coroutines/thread_id_type.hpp

Defined in header hpx/coroutines/thread_id_type.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<>

struct `std`::**hash**<::*hpx*::*threads*::*thread_id*>

Public Functions

inline *std*::size_t **operator()**(::*hpx*::*threads*::*thread_id* const &v) const noexcept

template<>

struct *std*::**hash**<::*hpx*::*threads*::*thread_id_ref*>

Public Functions

inline *std*::size_t **operator()**(::*hpx*::*threads*::*thread_id_ref* const &v) const noexcept

namespace **hpx**

namespace **threads**

Enums

enum class **thread_id_addref**

Values:

enumerator **yes**

enumerator **no**

Variables

constexpr const *thread_id* **invalid_thread_id**

struct **thread_id**

Public Functions

constexpr **thread_id**() noexcept = default

thread_id(*thread_id* const&) = default

thread_id &**operator=**(*thread_id* const&) = default

~thread_id() = default

inline constexpr **thread_id**(*thread_id* &&rhs) noexcept

inline constexpr *thread_id* &**operator=**(*thread_id* &&rhs) noexcept

inline explicit constexpr **thread_id**(*thread_id_repr* thrd) noexcept inline constexpr *thread_id* &**operator=**(*thread_id_repr* rhs) noexcept

inline explicit constexpr **operator bool**() const noexcept

inline constexpr *thread_id_repr* **get**() const noexcept

inline constexpr void **reset**() noexcept

Private Types

using **thread_id_repr** = void*

Private Members

thread_id_repr **thrd_** = nullptr

Friends

inline friend constexpr friend bool operator== (std::nullptr_t, thread_id const &rhs) noexcept

inline friend constexpr friend bool operator!= (std::nullptr_t, thread_id const &rhs) noexcept

inline friend constexpr friend bool operator== (thread_id const &lhs, std::nullptr_t) noexcept

inline friend constexpr friend bool operator!= (thread_id const &lhs, std::nullptr_t) noexcept

inline friend constexpr friend bool operator== (thread_id const &lhs, thread_id const &rhs) noexcept

inline friend constexpr friend bool operator!= (thread_id const &lhs, thread_id const &rhs) noexcept

inline friend constexpr friend bool operator< (thread_id const &lhs, thread_id const &rhs) noexcept

inline friend constexpr friend bool operator> (thread_id const &lhs, thread_id const &rhs) noexcept

inline friend constexpr friend bool operator<= (thread_id const &lhs, thread_id const &rhs) noexcept

inline friend constexpr friend bool operator>= (thread_id const &lhs, thread_id const &rhs) noexcept

friend *std*::ostream &**operator<<**(*std*::ostream &os, *thread_id* const &id)

friend void **format_value**(*std*::ostream &os, *std*::string_view spec, *thread_id* const &id)

struct **thread_id_ref**

Public Types

using **thread_repr** = detail::thread_data_reference_counting

Public Functions

thread_id_ref() noexcept = default

thread_id_ref(*thread_id_ref* const&) = default

thread_id_ref &**operator=**(*thread_id_ref* const&) = default

thread_id_ref(*thread_id_ref* &&rhs) noexcept = default

thread_id_ref &**operator=**(*thread_id_ref* &&rhs) noexcept = default

~thread_id_ref() = default

inline explicit **thread_id_ref**(*thread_id_repr* const &thrd) noexcept

inline explicit **thread_id_ref**(*thread_id_repr* &&thrd) noexcept

inline *thread_id_ref* &**operator=**(*thread_id_repr* const &rhs) noexcept

inline *thread_id_ref* &**operator=**(*thread_id_repr* &&rhs) noexcept

inline explicit **thread_id_ref**(*thread_repr* *thrd, *thread_id_addref* addref = *thread_id_addref*::*yes*) noexcept

inline *thread_id_ref* &**operator=**(*thread_repr* *rhs) noexcept

inline **thread_id_ref**(*thread_id* const &noref)

inline **thread_id_ref**(*thread_id* &&noref) noexcept

inline *thread_id_ref* &**operator=**(*thread_id* const &noref)

inline *thread_id_ref* &**operator=**(*thread_id* &&noref) noexcept

inline explicit constexpr **operator bool**() const noexcept

inline constexpr *thread_id* **noref**() const noexcept

inline constexpr *thread_id_repr* &**get**() & noexcept

inline *thread_id_repr* &&**get**() && noexcept inline constexpr *thread_id_repr* const &**get**() const & noexcept

inline void **reset**() noexcept

inline void **reset**(*thread_repr* *thrd, bool add_ref = true) noexcept

inline constexpr *thread_repr* ***detach**() noexcept

Private Types

using **thread_id_repr** = *hpx*::intrusive_ptr<detail::thread_data_reference_counting>

Private Members

thread_id_repr **thrd_**

Friends

inline friend constexpr friend bool operator== (std::nullptr_t, thread_id_ref const &rhs) noexcept

inline friend constexpr friend bool operator!= (std::nullptr_t, thread_id_ref const &rhs) noexcept

inline friend constexpr friend bool operator== (thread_id_ref const &lhs, std::nullptr_t) noexcept

inline friend constexpr friend bool operator!= (thread_id_ref const &lhs, std::nullptr_t) noexcept

inline friend constexpr friend bool operator== (thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept

inline friend constexpr friend bool operator!= (thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept

inline friend constexpr friend bool operator< (thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept

inline friend constexpr friend bool operator> (thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept

inline friend constexpr friend bool operator<= (thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept

inline friend constexpr friend bool operator>= (thread_id_ref const &lhs, thread_id_ref const &rhs) noexcept

friend *std*::ostream &**operator<<**(*std*::ostream &os, *thread_id_ref* const &id)

friend void **format_value**(*std*::ostream &os, *std*::string_view spec, *thread_id_ref* const &id)

namespace **std**

template<> thread_id >

Public Functions

inline *std*::size_t **operator()**(::*hpx*::*threads*::*thread_id* const &v) const noexcept

template<> thread_id_ref >

Public Functions

inline *std*::size_t **operator()**(::*hpx*::*threads*::*thread_id_ref* const &v) const noexcept

datastructures

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::any_nonser. hpx::bad_any_cast, hpx::unique_any_nonser, hpx::any_cast, hpx::make_any_nonser, hpx::make_unique_any_nonser

Defined in header hpx/any.hpp[686].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<>

class *hpx*::*util*::**basic_any**<void, void, void, *std*::true_type>

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* const &x)

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>

[686] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/any.hpp

inline explicit **basic_any**(T &&x, std::enable_if_t<std::is_copy_constructible_v<std::decay_t<T>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = std::enable_if_t<std::is_constructible_v<std::decay_t<T>, Ts...> && std::is_copy_constructible_v<std::decay_t<T>>>>
inline explicit **basic_any**(std::in_place_type_t<T>, Ts&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = std::enable_if_t<std::is_constructible_v<std::decay_t<T>, Ts...> && std::is_copy_constructible_v<std::decay_t<T>>>>
inline explicit **basic_any**(std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

inline **~basic_any**()

inline basic_any &**operator=**(basic_any const &x)

inline basic_any &**operator=**(basic_any &&rhs) noexcept

template<typename **T**, typename **Enable** = std::enable_if_t<!std::is_same_v<basic_any, std::decay_t<T>> && std::is_copy_constructible_v<std::decay_t<T>>>>
inline basic_any &**operator=**(T &&rhs)

inline basic_any &**swap**(basic_any &x) noexcept

inline std::type_info const &**type**() const

template<typename **T**>
inline T const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Functions

inline basic_any &**assign**(basic_any const &x)

Private Members

detail::any::fxn_ptr_table<void, void, void, std::true_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

template<typename **Char**>

class *hpx*::*util*::**basic_any**<void, void, *Char*, *std*::true_type>

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* const &x)

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline explicit **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_copy_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* const &x)

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline *basic_any* &**operator=**(*T* &&rhs) noexcept

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Functions

inline *basic_any* &**assign**(*basic_any* const &x)

Private Members

detail::any::fxn_ptr_table<void, void, *Char*, *std*::true_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

template<>

class *hpx*::*util*::**basic_any**<void, void, void, *std*::false_type>

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline explicit **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_move_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** =
std::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> &&
std::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** =
std::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> &&
std::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

basic_any(*basic_any* const &x) = delete

basic_any &**operator=**(*basic_any* const &x) = delete

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>
&& *std*::is_move_constructible_v<*std*::decay_t<*T*>>>>

inline *basic_any* &**operator=**(*T* &&rhs)

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Members

detail::any::fxn_ptr_table<void, void, void, *std*::false_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

template<typename **Char**>

class *hpx*::*util*::**basic_any**<void, void, *Char*, *std*::false_type>

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline explicit **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_move_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

basic_any(*basic_any* const &x) = delete *basic_any* &**operator=**(*basic_any* const &x) = delete

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_move_constructible_v<*std*::decay_t<*T*>>>>>
inline *basic_any* &**operator=**(*T* &&rhs) noexcept

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Members

detail::any::fxn_ptr_table<void, void, *Char*, *std*::false_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

namespace **hpx**

Top level HPX namespace.

Typedefs

using **any_nonser** = *util*::*basic_any*<void, void, void, *std*::true_type>

using **unique_any_nonser** = *util*::*basic_any*<void, void, void, *std*::false_type>

Functions

template<typename **T**, typename ...**Ts**>
util::*basic_any*<void, void, void, *std*::true_type> **make_any_nonser**(*Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**>
util::*basic_any*<void, void, void, *std*::true_type> **make_any_nonser**(*std*::initializer_list<*U*> il, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
util::*basic_any*<void, void, void, *std*::false_type> **make_unique_any_nonser**(*Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**>
util::*basic_any*<void, void, void, *std*::false_type> **make_unique_any_nonser**(*std*::initializer_list<*U*> il, *Ts*&&... ts)

template<typename **T**>
util::*basic_any*<void, void, void, *std*::true_type> **make_any_nonser**(*T* &&t)

template<typename **T**>
util::*basic_any*<void, void, void, *std*::false_type> **make_unique_any_nonser**(*T* &&t)

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T ***any_cast**(*util*::*basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> *operand) noexcept

Performs type-safe access to the contained object.

Parameters **operand** – target any object

Returns If operand is not a null pointer, and the *typeid* of the requested *T* matches that of the contents of *operand*, a pointer to the value contained by *operand*, otherwise a null pointer.

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T const ***any_cast**(*util*::*basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> const *operand) noexcept

Performs type-safe access to the contained object.

Parameters **operand** – target any object

Returns If operand is not a null pointer, and the *typeid* of the requested *T* matches that of the contents of *operand*, a pointer to the value contained by *operand*, otherwise a null pointer.

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T **any_cast**(*util*::*basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> &operand)

Performs type-safe access to the contained object. Let *U* be *std::remove_cv_t<std::remove_reference_t<T>>* The program is ill-formed if *std::is_constructible_v<T, U&>* is false.

Parameters **operand** – target any object

Returns static_cast<T>(*std::any_cast<U>(&operand))

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T const &**any_cast**(*util*::*basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> const &operand)

Performs type-safe access to the contained object. Let *U* be *std::remove_cv_t<std::remove_reference_t<T>>* The program is ill-formed if *std::is_constructible_v<T, const U&>* is false.

Parameters **operand** – target any object

Returns static_cast<T>(*std::any_cast<U>(&operand))

struct **bad_any_cast** : public bad_cast

#include <any.hpp> Defines a type of object to be thrown by the value-returning forms of *hpx::any_cast* on failure.

Public Functions

inline **bad_any_cast**(*std*::type_info const &src, *std*::type_info const &dest)

Constructs a new *bad_any_cast* object with an implementation-defined null-terminated byte string which is accessible through *what()*.

inline char const ***what**() const noexcept override

Returns the explanatory string.

Note: Implementations are allowed but not required to override *what()*.

Returns Pointer to a null-terminated string with explanatory information. The string is suitable for conversion and display as a std::wstring. The pointer is guaranteed to be valid at least until the exception object from which it is obtained is destroyed, or until a non-const member function (e.g. copy assignment operator) on the exception object is called.

Public Members

char const ***from**

char const ***to**

namespace **util**

Typedefs

using **streamable_any_nonser** = *basic_any*<void, void, char, *std*::true_type>

using **streamable_wany_nonser** = *basic_any*<void, void, wchar_t, *std*::true_type>

using **streamable_unique_any_nonser** = *basic_any*<void, void, char, *std*::false_type>

using **streamable_unique_wany_nonser** = *basic_any*<void, void, wchar_t, *std*::false_type>

Functions

template<typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**, typename **Enable** = *std*::enable_if_t<!*std*::is_void_v<*Char*>>>
std::basic_istream<*Char*> &**operator>>**(*std*::basic_istream<*Char*> &i, *basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> &obj)

template<typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**, typename **Enable** = *std*::enable_if_t<!*std*::is_void_v<*Char*>>>
std::basic_ostream<*Char*> &**operator<<**(*std*::basic_ostream<*Char*> &o, *basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> const &obj)

template<typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
void **swap**(*basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> &lhs, *basic_any*<*IArch*, *OArch*, *Char*, *Copyable*> &rhs) noexcept

template<typename **T**, typename **Char**, typename ...**Ts**>
basic_any<void, void, *Char*, *std*::true_type> **make_streamable_any_nonser**(*Ts*&&... ts)

template<typename **T**, typename **Char**, typename **U**, typename ...**Ts**>
basic_any<void, void, *Char*, *std*::true_type> **make_streamable_any_nonser**(*std*::initializer_list<*U*> il, *Ts*&&... ts)

template<typename **T**, typename **Char**, typename ...**Ts**>
basic_any<void, void, *Char*, *std*::false_type> **make_streamable_unique_any_nonser**(*Ts*&&... ts)

template<typename **T**, typename **Char**, typename **U**, typename ...**Ts**>
basic_any<void, void, *Char*, *std*::false_type> **make_streamable_unique_any_nonser**(*std*::initializer_list<*U*> il, *Ts*&&... ts)

template<typename **T**, typename **Char**>
basic_any<void, void, *Char*, *std*::true_type> **make_streamable_any_nonser**(*T* &&t)

template<typename **T**, typename **Char**>
basic_any<void, void, *Char*, *std*::false_type> **make_streamable_unique_any_nonser**(*T* &&t)

template<typename **IArch**, typename **OArch**, typename **Char** = char, typename **Copyable** = *std*::true_type>
class **basic_any**

template<typename Char> false_type >

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline explicit **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_move_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

basic_any(*basic_any* const &x) = delete

basic_any &**operator=**(*basic_any* const &x) = delete

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_move_constructible_v<*std*::decay_t<*T*>>>>
inline *basic_any* &**operator=**(*T* &&rhs) noexcept

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Members

detail::any::fxn_ptr_table<void, void, Char, *std*::false_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

template<typename Char> true_type >

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* const &x)

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline explicit **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_copy_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* const &x)

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline *basic_any* &**operator=**(*T* &&rhs) noexcept

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Functions

inline *basic_any* &**assign**(*basic_any* const &x)

Private Members

detail::any::fxn_ptr_table<void, void, Char, *std*::true_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

template<> false_type >

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline explicit **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_move_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

basic_any(*basic_any* const &x) = delete

basic_any &**operator=**(*basic_any* const &x) = delete

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_move_constructible_v<*std*::decay_t<*T*>>>>
inline *basic_any* &**operator=**(*T* &&rhs)

inline *basic_any* &**swap**(*basic_any* &x) noexcept inline std::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Members

detail::any::fxn_ptr_table<void, void, void, std::false_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, std::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, std::false_type, *Ts*&&... ts)

template<> true_type >

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* const &x)

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = std::enable_if_t<!std::is_same_v<*basic_any*, std::decay_t<*T*>>>>
inline explicit **basic_any**(*T* &&x, std::enable_if_t<std::is_copy_constructible_v<std::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = std::enable_if_t<std::is_constructible_v<std::decay_t<*T*>, *Ts*...> && std::is_copy_constructible_v<std::decay_t<*T*>>>>
inline explicit **basic_any**(std::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = std::enable_if_t<std::is_constructible_v<std::decay_t<*T*>, *Ts*...> && std::is_copy_constructible_v<std::decay_t<*T*>>>>
inline explicit **basic_any**(std::in_place_type_t<*T*>, std::initializer_list<*U*> il, *Ts*&&... ts)

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* const &x)

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>>
inline *basic_any* &**operator=**(*T* &&rhs)

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

Private Functions

inline *basic_any* &**assign**(*basic_any* const &x)

Private Members

detail::any::fxn_ptr_table<void, void, void, *std*::true_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

hpx::ignore, hpx::tuple, hpx::tuple_size, hpx::tuple_element, hpx::make_tuple, hpx::tie, hpx::forward_as_tuple, hpx::tuple_cat, hpx::get

Defined in header hpx/tuple.hpp[687].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

[687] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/tuple.hpp

Functions

template<typename ...**Ts**>
constexpr *tuple*<*util*::decay_unwrap_t<*Ts*>...> **make_tuple**(*Ts*&&... ts)

Provides compile-time indexed access to the types of the elements of the tuple.

template<typename ...**Ts**>
constexpr *tuple*<*Ts*&&...> **forward_as_tuple**(*Ts*&&... ts)

Constructs a tuple of references to the arguments in args suitable for forwarding as an argument to a function. The tuple has rvalue reference data members when rvalues are used as arguments, and otherwise has lvalue reference data members.

Parameters `ts` – zero or more arguments to construct the tuple from

Returns *hpx::tuple* object created as if by

```
hpx::tuple<Ts&&...>(HPX_FORWARD(Ts, ts)...)
```

template<typename ...**Ts**>
constexpr *tuple*<*Ts*&...> **tie**(*Ts*&... ts)

Creates a tuple of lvalue references to its arguments or instances of *hpx::ignore*.

Parameters `ts` – zero or more lvalue arguments to construct the tuple from.

Returns *hpx::tuple* object containing lvalue references.

template<typename ...**Tuples**>
constexpr auto **tuple_cat**(*Tuples*&&... tuples)

Constructs a tuple that is a concatenation of all tuples in `tuples`. The behavior is undefined if any type in `std::decay_t<Tuples>...` is not a specialization of `hpx::tuple`. However, an implementation may choose to support types (such as `std::array` and `std::pair`) that follow the tuple-like protocol.

Parameters `tuples` – - zero or more tuples to concatenate

Returns *hpx::tuple* object composed of all elements of all argument tuples constructed from `hpx::get<Is>`(HPX_FORWARD(UTuple,t) for each individual element.

template<*std*::size_t **I**>
util::at_index<*I*, Ts...>::type &**get**() noexcept

Extracts the Ith element from the tuple. I must be an integer value in [0, sizeof…(Ts)).

template<*std*::size_t **I**>
util::at_index<*I*, Ts...>::type const &**get**() const noexcept

Extracts the Ith element from the tuple. I must be an integer value in [0, sizeof…(Ts)).

Variables

constexpr *hpx*::detail::ignore_type **ignore** = {}

An object of unspecified type such that any value can be assigned to it with no effect. Intended for use with *hpx::tie* when unpacking a *hpx::tuple*, as a placeholder for the arguments that are not used. While the behavior of *hpx::ignore* outside of *hpx::tie* is not formally specified, some code guides recommend using *hpx::ignore* to avoid warnings from unused return values of [[nodiscard]] functions.

template<typename ...**Ts**>

class **tuple**

#include <tuple.hpp> Class template *hpx::tuple* is a fixed-size collection of heterogeneous values. It is a generalization of *hpx::pair*. If *std::is_trivially_destructible<Ti>::value* is true for every *Ti* in *Ts*, the destructor of tuple is trivial.

Param Ts... the types of the elements that the tuple stores.

template<*std*::size_t **I**, typename **T**, typename **Enable** = void>

struct **tuple_element**

#include <tuple.hpp> Provides compile-time indexed access to the types of the elements of a tuple-like type.

The primary template is not defined. An explicit (full) or partial specialization is required to make a type tuple-like.

template<typename **T**>

struct **tuple_size**

#include <tuple.hpp> Provides access to the number of elements in a tuple-like type as a compile-time constant expression.

The primary template is not defined. An explicit (full) or partial specialization is required to make a type tuple-like.

hpx::any, hpx::make_any

Defined in header hpx/any.hpp[688].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **IArch**, typename **OArch**, typename **Char**>

class *hpx*::*util*::**basic_any**<*IArch*, *OArch*, *Char*, *std*::true_type>

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* const &x)

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_copy_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** =
std::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> &&
std::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** =
std::enable_if<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> &&
std::is_copy_constructible_v<*std*::decay_t<*T*>>>>

688 http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/any.hpp

inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* const &x)

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>>
inline *basic_any* &**operator=**(*T* &&rhs)

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

inline bool **equal_to**(*basic_any* const &rhs) const noexcept

Private Functions

inline *basic_any* &**assign**(*basic_any* const &x)

inline void **load**(*IArch* &ar, unsigned const version)

inline void **save**(*OArch* &ar, unsigned const version) const

Private Members

detail::any::fxn_ptr_table<*IArch*, *OArch*, *Char*, *std*::true_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

Friends

friend class hpx::serialization::access

namespace **hpx**

Top level HPX namespace.

Typedefs

using **any** = *util*::*basic_any*<*serialization*::input_archive, *serialization*::output_archive, char, *std*::true_type>

Functions

template<typename **T**, typename **Char**>
util::*basic_any*<*serialization*::input_archive, *serialization*::output_archive, *Char*> **make_any**(*T* &&t)

Constructs an any object containing an object of type *T*, passing the provided arguments to T's constructor. Equivalent to:

```
return std::any(std::in_place_type<T>, std::forward<Args>(args)...);
```

namespace **util**

Typedefs

using **wany** = *basic_any*<*serialization*::input_archive, *serialization*::output_archive, wchar_t, *std*::true_type>

Functions

template<typename **T**, typename **Char**, typename ...**Ts**>
basic_any<*serialization*::input_archive, *serialization*::output_archive, *Char*> **make_any**(*Ts*&&... ts)

template<typename **T**, typename **Char**, typename **U**, typename ...**Ts**>
basic_any<*serialization*::input_archive, *serialization*::output_archive, *Char*> **make_any**(*std*::initializer_list<*U*> il, *Ts*&&... ts)

template<typename IArch, typename OArch, typename Char> true_type >

Public Functions

inline constexpr **basic_any**() noexcept

inline **basic_any**(*basic_any* const &x)

inline **basic_any**(*basic_any* &&x) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>>>>
inline **basic_any**(*T* &&x, *std*::enable_if_t<*std*::is_copy_constructible_v<*std*::decay_t<*T*>>>* = nullptr)

template<typename **T**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *Ts*&&... ts)

template<typename **T**, typename **U**, typename ...**Ts**, typename **Enable** = *std*::enable_if<*std*::is_constructible_v<*std*::decay_t<*T*>, *Ts*...> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline explicit **basic_any**(*std*::in_place_type_t<*T*>, *std*::initializer_list<*U*> il, *Ts*&&... ts)

inline **~basic_any**()

inline *basic_any* &**operator=**(*basic_any* const &x)

inline *basic_any* &**operator=**(*basic_any* &&rhs) noexcept

template<typename **T**, typename **Enable** = *std*::enable_if<!*std*::is_same_v<*basic_any*, *std*::decay_t<*T*>> && *std*::is_copy_constructible_v<*std*::decay_t<*T*>>>>
inline *basic_any* &**operator=**(*T* &&rhs)

inline *basic_any* &**swap**(*basic_any* &x) noexcept

inline *std*::type_info const &**type**() const

template<typename **T**>
inline *T* const &**cast**() const

inline bool **has_value**() const noexcept

inline void **reset**()

inline bool **equal_to**(*basic_any* const &rhs) const noexcept

Private Functions

inline *basic_any* &**assign**(*basic_any* const &x)

inline void **load**(IArch &ar, unsigned const version)

inline void **save**(OArch &ar, unsigned const version) const

Private Members

detail::any::fxn_ptr_table<IArch, OArch, Char, *std*::true_type> ***table**

void ***object**

Private Static Functions

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::true_type, *Ts*&&... ts)

template<typename **T**, typename ...**Ts**>
static inline void **new_object**(void *&object, *std*::false_type, *Ts*&&... ts)

Friends

friend class hpx::serialization::access

struct **hash_any**

Public Functions

template<typename **Char**>
std::size_t **operator()**(*basic_any*<*serialization*::input_archive, *serialization*::output_archive, *Char*, *std*::true_type> const &elem) const

debugging

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/debugging/print.hpp

Defined in header hpx/debugging/print.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_DP_LAZY(Expr, printer)

errors

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/errors/error.hpp

Defined in header hpx/errors/error.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_ERROR_UNSCOPED_ENUM_DEPRECATION_MSG

namespace **hpx**

Enums

enum class **error** : *std*::int16_t

Possible error conditions.

This enumeration lists all possible error conditions which can be reported from any of the API functions.

Values:

enumerator **success**

The operation was successful.

enumerator **no_success**

The operation did failed, but not in an unexpected manner.

enumerator **not_implemented**

The operation is not implemented.

enumerator **out_of_memory**

The operation caused an out of memory condition.

enumerator **bad_action_code**

enumerator **bad_component_type**

The specified component type is not known or otherwise invalid.

enumerator **network_error**

A generic network error occurred.

enumerator **version_too_new**

The version of the network representation for this object is too new.

enumerator **version_too_old**

The version of the network representation for this object is too old.

enumerator **version_unknown**

The version of the network representation for this object is unknown.

enumerator **unknown_component_address**

enumerator **duplicate_component_address**

The given global id has already been registered.

enumerator **invalid_status**

The operation was executed in an invalid status.

enumerator **bad_parameter**

One of the supplied parameters is invalid.

enumerator **internal_server_error**

enumerator **service_unavailable**

enumerator **bad_request**

enumerator **repeated_request**

enumerator **lock_error**

enumerator **duplicate_console**

There is more than one console locality.

enumerator **no_registered_console**

There is no registered console locality available.

enumerator **startup_timed_out**

enumerator **uninitialized_value**

enumerator **bad_response_type**

enumerator **deadlock**

enumerator **assertion_failure**

enumerator **null_thread_id**

Attempt to invoke a API function from a non-HPX thread.

enumerator **invalid_data**

enumerator **yield_aborted**

The yield operation was aborted.

enumerator **dynamic_link_failure**

enumerator **commandline_option_error**

One of the options given on the command line is erroneous.

enumerator **serialization_error**

There was an error during serialization of this object.

enumerator **unhandled_exception**

An unhandled exception has been caught.

enumerator **kernel_error**

The OS kernel reported an error.

enumerator **broken_task**

The task associated with this future object is not available anymore.

enumerator **task_moved**

The task associated with this future object has been moved.

enumerator **task_already_started**

The task associated with this future object has already been started.

enumerator **future_already_retrieved**

The future object has already been retrieved.

enumerator **promise_already_satisfied**

The value for this future object has already been set.

enumerator **future_does_not_support_cancellation**

The future object does not support cancellation.

enumerator **future_can_not_be_cancelled**

The future can't be canceled at this time.

enumerator **no_state**

The future object has no valid shared state.

enumerator **broken_promise**

The promise has been deleted.

enumerator **thread_resource_error**

enumerator **future_cancelled**

enumerator **thread_cancelled**

enumerator **thread_not_interruptable**

enumerator **duplicate_component_id**

The component type has already been registered.

enumerator **unknown_error**

An unknown error occurred.

enumerator **bad_plugin_type**

The specified plugin type is not known or otherwise invalid.

enumerator **filesystem_error**

The specified file does not exist or other filesystem related error.

enumerator **bad_function_call**

equivalent of std::bad_function_call

enumerator **task_canceled_exception**

parallel::task_canceled_exception

enumerator **task_block_not_active**

task_region is not active

enumerator **out_of_range**

Equivalent to std::out_of_range.

enumerator **length_error**

Equivalent to std::length_error.

enumerator **migration_needs_retry**

migration failed because of global race, retry

Functions

inline constexpr bool **operator==**(int lhs, *error* rhs) noexcept

inline constexpr bool **operator==**(*error* lhs, int rhs) noexcept

inline constexpr bool **operator!=**(int lhs, *error* rhs) noexcept

inline constexpr bool **operator!=**(*error* lhs, int rhs) noexcept

inline constexpr bool **operator<**(int lhs, *error* rhs) noexcept

inline constexpr bool **operator>=**(int lhs, *error* rhs) noexcept

inline constexpr int **operator&**(*error* lhs, *error* rhs) noexcept

inline constexpr int **operator&**(int lhs, *error* rhs) noexcept

inline constexpr int **operator|=**(int &lhs, *error* rhs) noexcept

char const ***get_error_name**(*error* e) noexcept

Variables

constexpr *error* **success** = *error*::*success*

constexpr *error* **no_success** = *error*::*no_success*

constexpr *error* **not_implemented** = *error*::*not_implemented*

constexpr *error* **out_of_memory** = *error*::*out_of_memory*

constexpr *error* **bad_action_code** = *error*::*bad_action_code*

constexpr *error* **bad_component_type** = *error*::*bad_component_type*

constexpr *error* **network_error** = *error*::*network_error*

constexpr *error* **version_too_new** = *error*::*version_too_new*

constexpr *error* **version_too_old** = *error*::*version_too_old*

constexpr *error* **version_unknown** = *error*::*version_unknown*

constexpr *error* **unknown_component_address** = *error*::*unknown_component_address*

constexpr *error* **duplicate_component_address** = *error*::*duplicate_component_address* constexpr *error* **invalid_status** = *error*::*invalid_status*

constexpr *error* **bad_parameter** = *error*::*bad_parameter*

constexpr *error* **internal_server_error** = *error*::*internal_server_error*

constexpr *error* **service_unavailable** = *error*::*service_unavailable*

constexpr *error* **bad_request** = *error*::*bad_request*

constexpr *error* **repeated_request** = *error*::*repeated_request*

constexpr *error* **lock_error** = *error*::*lock_error*

constexpr *error* **duplicate_console** = *error*::*duplicate_console*

constexpr *error* **no_registered_console** = *error*::*no_registered_console*

constexpr *error* **startup_timed_out** = *error*::*startup_timed_out*

constexpr *error* **uninitialized_value** = *error*::*uninitialized_value*

constexpr *error* **bad_response_type** = *error*::*bad_response_type*

constexpr *error* **deadlock** = *error*::*deadlock*

constexpr *error* **assertion_failure** = *error*::*assertion_failure*

constexpr *error* **null_thread_id** = *error*::*null_thread_id*

constexpr *error* **invalid_data** = *error*::*invalid_data*

constexpr *error* **yield_aborted** = *error*::*yield_aborted*

constexpr *error* **dynamic_link_failure** = *error*::*dynamic_link_failure*

constexpr *error* **commandline_option_error** = *error*::*commandline_option_error*

constexpr *error* **serialization_error** = *error*::*serialization_error*

constexpr *error* **unhandled_exception** = *error*::*unhandled_exception* constexpr *error* **kernel_error** = *error*::*kernel_error*

constexpr *error* **broken_task** = *error*::*broken_task*

constexpr *error* **task_moved** = *error*::*task_moved*

constexpr *error* **task_already_started** = *error*::*task_already_started*

constexpr *error* **future_already_retrieved** = *error*::*future_already_retrieved*

constexpr *error* **promise_already_satisfied** = *error*::*promise_already_satisfied*

constexpr *error* **future_does_not_support_cancellation** = *error*::*future_does_not_support_cancellation*

constexpr *error* **future_can_not_be_cancelled** = *error*::*future_can_not_be_cancelled*

constexpr *error* **no_state** = *error*::*no_state*

constexpr *error* **broken_promise** = *error*::*broken_promise*

constexpr *error* **thread_resource_error** = *error*::*thread_resource_error*

constexpr *error* **future_cancelled** = *error*::*future_cancelled*

constexpr *error* **thread_cancelled** = *error*::*thread_cancelled*

constexpr *error* **thread_not_interruptable** = *error*::*thread_not_interruptable*

constexpr *error* **duplicate_component_id** = *error*::*duplicate_component_id*

constexpr *error* **unknown_error** = *error*::*unknown_error*

constexpr *error* **bad_plugin_type** = *error*::*bad_plugin_type*

constexpr *error* **filesystem_error** = *error*::*filesystem_error*

constexpr *error* **bad_function_call** = *error*::*bad_function_call*

constexpr *error* **task_canceled_exception** = *error*::*task_canceled_exception*

constexpr *error* **task_block_not_active** = *error*::*task_block_not_active* constexpr *error* **out_of_range** = *error*::*out_of_range*

constexpr *error* **length_error** = *error*::*length_error*

constexpr *error* **migration_needs_retry** = *error*::*migration_needs_retry*

hpx::error_code

Defined in header hpx/system_error.hpp[689].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Unnamed Group

inline *error_code* **make_error_code**(*error* e, *throwmode* mode = *throwmode*::*plain*)

Returns a new *error_code* constructed from the given parameters.

inline *error_code* **make_error_code**(*error* e, char const *func, char const *file, long line, *throwmode* mode = *throwmode*::*plain*)

inline *error_code* **make_error_code**(*error* e, char const *msg, *throwmode* mode = *throwmode*::*plain*)

Returns error_code(e, msg, mode).

inline *error_code* **make_error_code**(*error* e, char const *msg, char const *func, char const *file, long line, *throwmode* mode = *throwmode*::*plain*)

inline *error_code* **make_error_code**(*error* e, *std*::string const &msg, *throwmode* mode = *throwmode*::*plain*)

Returns error_code(e, msg, mode).

inline *error_code* **make_error_code**(*error* e, *std*::string const &msg, char const *func, char const *file, long line, *throwmode* mode = *throwmode*::*plain*)

inline *error_code* **make_error_code**(*std*::exception_ptr const &e)

Functions

std::error_category const &**get_hpx_category**() noexcept

Returns generic HPX error category used for new errors.

std::error_category const &**get_hpx_rethrow_category**() noexcept

Returns generic HPX error category used for errors re-thrown after the exception has been de-serialized.

inline *error_code* **make_success_code**(*throwmode* mode = *throwmode*::*plain*)

Returns *error_code*(hpx::error::success, "success", mode).

[689] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/system_error.hpp

class **error_code** : public *error_code*

#include <error_code.hpp> A *hpx::error_code* represents an arbitrary error condition.

The class *hpx::error_code* describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces.

Note: Class *hpx::error_code* is an adjunct to error reporting by exception

---

Public Functions

inline explicit **error_code**(*throwmode* mode = *throwmode*::*plain*)

Construct an object of type *error_code*.

Parameters mode – The parameter `mode` specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

Throws `nothing` –

explicit **error_code**(*error* e, *throwmode* mode = *throwmode*::*plain*)

Construct an object of type *error_code*.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **mode** – The parameter `mode` specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

Throws `nothing` –

error_code(*error* e, char const *func, char const *file, long line, *throwmode* mode = *throwmode*::*plain*)

Construct an object of type *error_code*.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **func** – The name of the function where the error was raised.
- **file** – The file name of the code where the error was raised.
- **line** – The line number of the code line where the error was raised.
- **mode** – The parameter `mode` specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

Throws `nothing` –

error_code(*error* e, char const *msg, *throwmode* mode = *throwmode*::*plain*)

Construct an object of type *error_code*.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **msg** – The parameter `msg` holds the error message the new exception should encapsulate.
- **mode** – The parameter `mode` specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

Throws `std::bad_alloc` – (if allocation of a copy of the passed string fails).

error_code(*error* e, char const *msg, char const *func, char const *file, long line, *throwmode* mode = *throwmode*::*plain*)

Construct an object of type *error_code*.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **msg** – The parameter `msg` holds the error message the new exception should encapsulate.
- **func** – The name of the function where the error was raised.
- **file** – The file name of the code where the error was raised.
- **line** – The line number of the code line where the error was raised.
- **mode** – The parameter `mode` specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

Throws `std::bad_alloc` – (if allocation of a copy of the passed string fails).

error_code(*error* e, *std*::string const &msg, *throwmode* mode = *throwmode*::*plain*)

Construct an object of type *error_code*.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **msg** – The parameter `msg` holds the error message the new exception should encapsulate.
- **mode** – The parameter `mode` specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

Throws `std::bad_alloc` – (if allocation of a copy of the passed string fails).

error_code(*error* e, *std*::string const &msg, char const *func, char const *file, long line, *throwmode* mode = *throwmode*::*plain*)

Construct an object of type *error_code*.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **msg** – The parameter `msg` holds the error message the new exception should encapsulate.
- **func** – The name of the function where the error was raised.
- **file** – The file name of the code where the error was raised.
- **line** – The line number of the code line where the error was raised.
- **mode** – The parameter `mode` specifies whether the constructed *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

Throws `std::bad_alloc` – (if allocation of a copy of the passed string fails).

std::string **get_message**() const

Return a reference to the error message stored in the *hpx::error_code*.

Throws `nothing` –

inline void **clear**()

Clear this *error_code* object. The postconditions of invoking this method are.

- value() == hpx::error::success and category() == *hpx::get_hpx_category()*

error_code(*error_code* const &rhs)

Copy constructor for *error_code*

Note: This function maintains the error category of the left hand side if the right hand side is a success code.

error_code &**operator=**(*error_code* const &rhs)

Assignment operator for *error_code*

Note: This function maintains the error category of the left hand side if the right hand side is a success code.

Private Functions

error_code(int err, *hpx*::*exception* const &e)

explicit **error_code**(*std*::exception_ptr const &e)

Private Members

std::exception_ptr **exception_**

Friends

friend class exception

friend *error_code* **make_error_code**(*std*::exception_ptr const&)

hpx::exception

Defined in header hpx/exception.hpp[690].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Typedefs

using **custom_exception_info_handler_type** = *std*::function<*hpx*::exception_info(*std*::string const&, *std*::string const&, long, *std*::string const&)>

using **pre_exception_handler_type** = *std*::function<void()>

Functions

void **set_custom_exception_info_handler**(*custom_exception_info_handler_type* f)

void **set_pre_exception_handler**(*pre_exception_handler_type* f)

[690] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/exception.hpp

std::string **get_error_what**(exception_info const &xi)

Return the error message of the thrown exception.

The function *hpx::get_error_what* can be used to extract the diagnostic information element representing the error message as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, hpx::get_error_function_name(), hpx::get_error_file_name(), hpx::get_error_line_number(), *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, hpx::get_error() *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters `xi` – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, hpx::error_code, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns The error message stored in the exception If the exception instance does not hold this information, the function will return an empty string.

error **get_error**(*hpx*::*exception* const &e)

Return the error code value of the exception thrown.

The function *hpx::get_error* can be used to extract the diagnostic information element representing the error value code as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, hpx::get_error_function_name(), hpx::get_error_file_name(), hpx::get_error_line_number(), *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, hpx::get_error_what(), *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters `e` – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: hpx::exception, hpx::error_code, or *std::exception_ptr*.

Throws `nothing` –

Returns The error value code of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return *hpx::naming::invalid_locality_id*.

error **get_error**(*hpx*::*error_code* const &e)

std::string **get_error_function_name**(*hpx*::exception_info const &xi)

Return the function name from which the exception was thrown.

The function *hpx::get_error_function_name* can be used to extract the diagnostic information element representing the name of the function as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()* hpx::get_error_file_name(), hpx::get_error_line_number(), *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, hpx::get_error(), *hpx::get_error_backtrace()*, *hpx::get_error_env()*, hpx::get_error_what(), *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters **xi** – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, hpx::error_code, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns The name of the function from which the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

std::string **get_error_file_name**(*hpx*::exception_info const &xi)

Return the (source code) file name of the function from which the exception was thrown.

The function *hpx::get_error_file_name* can be used to extract the diagnostic information element representing the name of the source file as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, hpx::get_error_function_name(), hpx::get_error_line_number(), *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, hpx::get_error(), *hpx::get_error_backtrace()*, *hpx::get_error_env()*, hpx::get_error_what(), *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters **xi** – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, hpx::error_code, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns The name of the source file of the function from which the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

long **get_error_line_number**(*hpx*::exception_info const &xi)

Return the line number in the (source code) file of the function from which the exception was thrown.

The function *hpx::get_error_line_number* can be used to extract the diagnostic information element representing the line number as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, hpx::get_error_function_name(), hpx::get_error_file_name() *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, hpx::get_error(), *hpx::get_error_backtrace()*, *hpx::get_error_env()*, hpx::get_error_what(), *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters **xi** – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, hpx::error_code, *std::exception*, or *std::exception_ptr*.

Throws `nothing` –

Returns The line number of the place where the exception was thrown. If the exception instance does not hold this information, the function will return -1.

class **exception** : public system_error

#include <exception.hpp> A *hpx::exception* is the main exception type used by HPX to report errors.

The *hpx::exception* type is the main exception type used by HPX to report errors. Any exceptions thrown by functions in the HPX library are either of this type or of a type derived from it. This implies that it is always safe to use this type only in catch statements guarding HPX library calls.

Subclassed by *hpx::exception_list*

Public Functions

explicit **exception**(*error* e = *hpx*::*error*::*success*)

Construct a *hpx::exception* from a *hpx::error*.

Parameters **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.

explicit **exception**(*std*::system_error const &e)

Construct a *hpx::exception* from a boost::system_error.

explicit **exception**(*std*::error_code const &e)

Construct a *hpx::exception* from a boost::system::error_code (this is new for Boost V1.69). This constructor is required to compensate for the changes introduced as a resolution to LWG3162 (https://cplusplus.github.io/LWG/issue3162).

exception(*error* e, char const *msg, *throwmode* mode = *throwmode*::*plain*)

Construct a *hpx::exception* from a *hpx::error* and an error message.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **msg** – The parameter `msg` holds the error message the new exception should encapsulate.
- **mode** – The parameter `mode` specifies whether the returned *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

exception(*error* e, *std*::string const &msg, *throwmode* mode = *throwmode*::*plain*)

Construct a *hpx::exception* from a *hpx::error* and an error message.

Parameters

- **e** – The parameter `e` holds the hpx::error code the new exception should encapsulate.
- **msg** – The parameter `msg` holds the error message the new exception should encapsulate.
- **mode** – The parameter `mode` specifies whether the returned *hpx::error_code* belongs to the error category *hpx_category* (if mode is *plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

~exception() override

Destruct a *hpx::exception*

Throws `nothing` –

error **get_error**() const noexcept

The function get_error() returns the hpx::error code stored in the referenced instance of a *hpx::exception*. It returns the hpx::error code this exception instance was constructed from.

Throws nothing –

error_code **get_error_code**(*throwmode* mode = *throwmode*::*plain*) const noexcept

The function get_error_code() returns a *hpx::error_code* which represents the same error condition as this *hpx::exception* instance.

Parameters **mode** – The parameter mode specifies whether the returned *hpx::error_code* belongs to the error category *hpx_category* (if mode is *throwmode::plain*, this is the default) or to the category *hpx_category_rethrow* (if mode is *rethrow*).

struct **thread_interrupted** : public *exception*

#include <exception.hpp> A *hpx::thread_interrupted* is the exception type used by HPX to interrupt a running HPX thread.

The hpx::thread_interrupted type is the exception type used by HPX to interrupt a running thread.

A running thread can be interrupted by invoking the interrupt() member function of the corresponding hpx::thread object. When the interrupted thread next executes one of the specified interruption points (or if it is currently blocked whilst executing one) with interruption enabled, then a *hpx::thread_interrupted* exception will be thrown in the interrupted thread. If not caught, this will cause the execution of the interrupted thread to terminate. As with any other exception, the stack will be unwound, and destructors for objects of automatic storage duration will be executed.

If a thread wishes to avoid being interrupted, it can create an instance of *hpx::this_thread::disable_interruption*. Objects of this class disable interruption for the thread that created them on construction, and restore the interruption state to whatever it was before on destruction.

```
void f()
{
    // interruption enabled here
    {
        hpx::this_thread::disable_interruption di;
        // interruption disabled
        {
            hpx::this_thread::disable_interruption di2;
            // interruption still disabled
        } // di2 destroyed, interruption state restored
        // interruption still disabled
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

The effects of an instance of *hpx::this_thread::disable_interruption* can be temporarily reversed by constructing an instance of *hpx::this_thread::restore_interruption*, passing in the *hpx::this_thread::disable_interruption* object in question. This will restore the interruption state to what it was when the *hpx::this_thread::disable_interruption* object was constructed, and then disable interruption again when the *hpx::this_thread::restore_interruption* object is destroyed.

```
void g()
{
```

(continues on next page)

(continued from previous page)

```
    // interruption enabled here
    {
        hpx::this_thread::disable_interruption di;
        // interruption disabled
        {
            hpx::this_thread::restore_interruption ri(di);
            // interruption now enabled
        } // ri destroyed, interruption disable again
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

At any point, the interruption state for the current thread can be queried by calling *hpx::this_thread::interruption_enabled()*.

hpx/errors/exception_fwd.hpp

Defined in header hpx/errors/exception_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_THROWMODE_UNSCOPED_ENUM_DEPRECATION_MSG

namespace **hpx**

Enums

enum class **throwmode** : *std*::uint8_t

Encode error category for new *error_code*.

Values:

enumerator **plain**

enumerator **rethrow**

enumerator **lightweight**

Functions

constexpr bool **operator&**(*throwmode* lhs, *throwmode* rhs) noexcept

Variables

constexpr *throwmode* **plain** = *throwmode*::*plain*

constexpr *throwmode* **rethrow** = *throwmode*::*rethrow*

constexpr *throwmode* **lightweight** = *throwmode*::*lightweight*

constexpr *throwmode* **lightweight_rethrow** = *throwmode*::lightweight_rethrow

error_code **throws**

Predefined *error_code* object used as "throw on error" tag.

The predefined *hpx::error_code* object *hpx::throws* is supplied for use as a "throw on error" tag.

Functions that specify an argument in the form '*error_code*& ec=throws' (with appropriate namespace qualifiers), have the following error handling semantics:

If &ec != &throws and an error occurred: ec.value() returns the implementation specific error number for the particular error that occurred and ec.category() returns the error_category for ec.value().

If &ec != &throws and an error did not occur, ec.clear().

If an error occurs and &ec == &throws, the function throws an exception of type hpx::exception or of a type derived from it. The exception's *get_errorcode()* member function returns a reference to an hpx::error_code object with the behavior as specified above.

hpx/errors/exception_list.hpp

Defined in header hpx/errors/exception_list.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

class **exception_list** : public *hpx*::*exception*

#include <exception_list.hpp> The class *exception_list* is a container of exception_ptr objects parallel algorithms may use to communicate uncaught exceptions encountered during parallel execution to the caller of the algorithm

The type exception_list::const_iterator fulfills the requirements of a forward iterator.

Public Types

using **iterator** = exception_list_type::const_iterator

bidirectional iterator

Public Functions

inline *std*::size_t **size**() const noexcept

The number of exception_ptr objects contained within the *exception_list*.

Note: Complexity: Constant time.

inline exception_list_type::const_iterator **begin**() const noexcept

An iterator referring to the first exception_ptr object contained within the *exception_list*.

inline exception_list_type::const_iterator **end**() const noexcept

An iterator which is the past-the-end value for the *exception_list*.

HPX_THROW_EXCEPTION, HPX_THROWS_IF

Defined in header hpx/exception.hpp[691].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_THROW_EXCEPTION(errcode, f, ...)

Throw a *hpx::exception* initialized from the given parameters.

The macro *HPX_THROW_EXCEPTION* can be used to throw a *hpx::exception*. The purpose of this macro is to prepend the source file name and line number of the position where the exception is thrown to the error message. Moreover, this associates additional diagnostic information with the exception, such as file name and line number, locality id and thread id, and stack backtrace from the point where the exception was thrown.

The parameter `errcode` holds the hpx::error code the new exception should encapsulate. The parameter `f` is expected to hold the name of the function exception is thrown from and the parameter `msg` holds the error message the new exception should encapsulate.

```
void raise_exception()
{
    // Throw a hpx::exception initialized from the given parameters.
    // Additionally associate with this exception some detailed
    // diagnostic information about the throw-site.
    HPX_THROW_EXCEPTION(hpx::error::no_success, "raise_exception",
       "simulated error");
}
```

[691] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/exception.hpp

Example:

HPX_THROWS_IF(ec, errcode, f, ...)

Either throw a *hpx::exception* or initialize hpx::error_code from the given parameters.

The macro *HPX_THROWS_IF* can be used to either throw a hpx::exception or to initialize a hpx::error_code from the given parameters. If &ec == &hpx::throws, the semantics of this macro are equivalent to *HPX_THROW_EXCEPTION*. If &ec != &hpx::throws, the hpx::error_code instance `ec` is initialized instead.

The parameter `errcode` holds the hpx::error code from which the new exception should be initialized. The parameter `f` is expected to hold the name of the function exception is thrown from and the parameter `msg` holds the error message the new exception should encapsulate.

namespace **hpx**

execution

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/execution/executors/adaptive_static_chunk_size.hpp

Defined in header hpx/execution/executors/adaptive_static_chunk_size.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

Typedefs

typedef *hpx*::*execution*::*experimental*::*adaptive_static_chunk_size* **instead**

namespace **experimental**

struct **adaptive_static_chunk_size**

#include <adaptive_static_chunk_size.hpp> Loop iterations are divided into pieces of size *chunk_size* and then assigned to threads. If *chunk_size* is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

Note: This executor parameters type is equivalent to OpenMP's STATIC scheduling directive.

Public Functions

adaptive_static_chunk_size() = default

Construct a adaptive_static_chunk_size executor parameters object

Note: By default the number of loop iterations is determined from the number of available cores and the overall number of loop iterations to schedule.

inline explicit constexpr **adaptive_static_chunk_size**(*std*::size_t chunk_size) noexcept

Construct a adaptive_static_chunk_size executor parameters object

Parameters **chunk_size** – [in] The optional chunk size to use as the number of loop iterations to run on a single thread.

hpx::execution::experimental::auto_chunk_size

Defined in header hpx/execution.hpp[692].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

struct **auto_chunk_size**

#include <auto_chunk_size.hpp> Loop iterations are divided into pieces and then assigned to threads. The number of loop iterations combined is determined based on measurements of how long the execution of 1% of the overall number of iterations takes. This executor parameters type makes sure that as many loop iterations are combined as necessary to run for the amount of time specified.

Public Functions

inline explicit constexpr **auto_chunk_size**(*std*::uint64_t num_iters_for_timing = 0) noexcept

Construct an auto_chunk_size executor parameters object

Note: Default constructed auto_chunk_size executor parameter types will use 80 microseconds as the minimal time for which any of the scheduled chunks should run.

inline explicit **auto_chunk_size**(*hpx*::*chrono*::steady_duration const &rel_time, *std*::uint64_t num_iters_for_timing = 0) noexcept

Construct an auto_chunk_size executor parameters object

Parameters

[692] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp

- **rel_time** – [in] The time duration to use as the minimum to decide how many loop iterations should be combined.
- **num_iters_for_timing** – [in] The number of iterations to use for the timing operation

hpx/execution/executors/default_parameters.hpp

Defined in header hpx/execution/executors/default_parameters.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

struct **default_parameters**

#include <default_parameters.hpp> Loop iterations are divided into pieces of size *chunk_size* and then assigned to threads. If *chunk_size* is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

Note: This executor parameters type is equivalent to OpenMP's STATIC scheduling directive.

Public Functions

default_parameters() = default

Construct a default_parameters executor parameters object

Note: By default the number of loop iterations is determined from the number of available cores and the overall number of loop iterations to schedule.

hpx::execution::experimental::dynamic_chunk_size

Defined in header hpx/execution.hpp[693].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

[693] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp

namespace **experimental**

struct **dynamic_chunk_size**

#include <dynamic_chunk_size.hpp> Loop iterations are divided into pieces of size *chunk_size* and then dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another If *chunk_size* is not specified, the default chunk size is 1.

Note: This executor parameters type is equivalent to OpenMP's DYNAMIC scheduling directive.

Public Functions

dynamic_chunk_size() = default

Construct an dynamic_chunk_size executor parameters object

Note: Default constructed dynamic_chunk_size executor parameter types will use a chunk size of '1'.

inline explicit constexpr **dynamic_chunk_size**(*std*::size_t chunk_size) noexcept

Construct a dynamic_chunk_size executor parameters object

Parameters chunk_size – [in] The optional chunk size to use as the number of loop iterations to schedule together. The default chunk size is 1.

hpx/execution/executors/execution.hpp

Defined in header hpx/execution/executors/execution.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

hpx/execution/executors/execution_information.hpp

Defined in header hpx/execution/executors/execution_information.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Variables

hpx::*parallel*::*execution*::*has_pending_closures_t* **has_pending_closures**

hpx::*parallel*::*execution*::*get_pu_mask_t* **get_pu_mask**

hpx::*parallel*::*execution*::*set_scheduler_mode_t* **set_scheduler_mode**

struct **get_pu_mask_t** : public *hpx*::*functional*::detail::tag_fallback<*get_pu_mask_t*>

#include <execution_information.hpp> Retrieve the bitmask describing the processing units the given thread is allowed to run on

All threads::executors invoke sched.get_pu_mask().

Note: If the executor does not support this operation, this call will always invoke hpx::threads::get_pu_mask()

Param exec [in] The executor object to use for querying the number of pending tasks.
Param topo [in] The topology object to use to extract the requested information.
Param thream_num [in] The sequence number of the thread to retrieve information for.

Private Functions

template<typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*get_pu_mask_t*, *Executor*&&, *threads*::*topology* &topo, *std*::size_t thread_num)

template<typename **Executor**>
inline decltype(auto) friend **tag_invoke**(*get_pu_mask_t*, *Executor* &&exec, *threads*::*topology* &topo, *std*::size_t thread_num)

struct **has_pending_closures_t** : public *hpx*::*functional*::detail::tag_fallback<*has_pending_closures_t*>

#include <execution_information.hpp> Retrieve whether this executor has operations pending or not.

Note: If the executor does not expose this information, this call will always return *false*

Param exec [in] The executor object to use to extract the requested information for.

Private Functions

template<typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*has_pending_closures_t*, *Executor*&&)

template<typename **Executor**>
inline decltype(auto) friend **tag_invoke**(*has_pending_closures_t*, *Executor* &&exec)

struct **set_scheduler_mode_t** : public *hpx*::*functional*::detail::tag_fallback<*set_scheduler_mode_t*>

#include <execution_information.hpp> Set various modes of operation on the scheduler underneath the given executor.

Note: This calls exec.set_scheduler_mode(mode) if it exists; otherwise it does nothing.

Param exec [in] The executor object to use.
Param mode [in] The new mode for the scheduler to pick up

Friends

template<typename **Executor**, typename **Mode**>
inline friend void **tag_fallback_invoke**(*set_scheduler_mode_t*, *Executor*&&, *Mode* const&)

template<typename **Executor**, typename **Mode**>
inline friend void **tag_invoke**(*set_scheduler_mode_t*, *Executor* &&exec, *Mode* const &mode)

hpx/execution/executors/execution_parameters.hpp

Defined in header hpx/execution/executors/execution_parameters.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Functions

template<typename ...**Params**>
constexpr *executor_parameters_join*<*Params*...>::type **join_executor_parameters**(*Params*&&... params)

template<typename **Param**>
constexpr *Param* &&**join_executor_parameters**(*Param* &&param) noexcept

template<typename ...**Params**>

struct **executor_parameters_join**

Public Types

using **type** = detail::executor_parameters<*std*::decay_t<*Params*>...>

template<typename **Param**>

struct **executor_parameters_join**<*Param*>

Public Types

using **type** = *Param*

hpx/execution/executors/execution_parameters_fwd.hpp

Defined in header hpx/execution/executors/execution_parameters_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<>

struct **is_scheduling_property**<*hpx*::*parallel*::*execution*::*with_processing_units_count_t*> : public true_type

namespace **hpx**

namespace **parallel**

namespace **execution**

Variables

constexpr struct *hpx*::*parallel*::*execution*::*null_parameters_t* **null_parameters**

hpx::*parallel*::*execution*::*get_chunk_size_t* **get_chunk_size**

hpx::*parallel*::*execution*::*measure_iteration_t* **measure_iteration**

hpx::*parallel*::*execution*::*maximal_number_of_chunks_t* **maximal_number_of_chunks**

hpx::*parallel*::*execution*::*reset_thread_distribution_t* **reset_thread_distribution**

hpx::*parallel*::*execution*::*processing_units_count_t* **processing_units_count**

hpx::*parallel*::*execution*::*with_processing_units_count_t* **with_processing_units_count** *hpx*::*parallel*::*execution*::*mark_begin_execution_t* **mark_begin_execution**

hpx::*parallel*::*execution*::*mark_end_of_scheduling_t* **mark_end_of_scheduling**

hpx::*parallel*::*execution*::*mark_end_execution_t* **mark_end_execution**

struct **get_chunk_size_t** : public *hpx*::*functional*::detail::tag_priority<*get_chunk_size_t*>

#include <execution_parameters_fwd.hpp> Return the number of invocations of the given function *f* which should be combined into a single task

Param params [in] The executor parameters object to use for determining the chunk size for the given number of tasks *num_tasks*.

Param exec [in] The executor object which will be used for scheduling of the loop iterations.

Param iteration_duration [in] The time one of the tasks require to be executed.

Param cores [in] The number of cores the number of chunks should be determined for.

Param num_tasks [in] The number of tasks the chunk size should be determined for

Return The size of the chunks (number of iterations per chunk) that should be used for parallel execution.

Private Functions

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*get_chunk_size_t*, *Parameters* &&params, *Executor* &&exec, *hpx*::*chrono*::steady_duration const &iteration_duration, *std*::size_t cores, *std*::size_t num_tasks)

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*get_chunk_size_t* tag, *Parameters* &&params, *Executor* &&exec, *std*::size_t cores, *std*::size_t num_tasks)

struct **mark_begin_execution_t** : public *hpx*::*functional*::detail::tag_priority<*mark_begin_execution_t*>

#include <execution_parameters_fwd.hpp> Mark the begin of a parallel algorithm execution

Note: This calls params.mark_begin_execution(exec) if it exists; otherwise it does nothing.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*mark_begin_execution_t*, *Parameters* &&params, *Executor* &&exec)

struct **mark_end_execution_t** : public *hpx*::*functional*::detail::tag_priority<*mark_end_execution_t*>

#include <execution_parameters_fwd.hpp> Mark the end of a parallel algorithm execution

Note: This calls params.mark_end_execution(exec) if it exists; otherwise it does nothing.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*mark_end_execution_t*, *Parameters* &&params, *Executor* &&exec)

struct **mark_end_of_scheduling_t** : public *hpx*::*functional*::detail::tag_priority<*mark_end_of_scheduling_t*>

#include <execution_parameters_fwd.hpp> Mark the end of scheduling tasks during parallel algorithm execution

Note: This calls params.mark_begin_execution(exec) if it exists; otherwise it does nothing.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*mark_end_of_scheduling_t*, *Parameters* &&params, *Executor* &&exec)

struct **maximal_number_of_chunks_t** : public *hpx*::*functional*::detail::tag_priority<*maximal_number_of_chunks_t*>

#include <execution_parameters_fwd.hpp> Return the largest reasonable number of chunks to create for a single algorithm invocation.

Param params [in] The executor parameters object to use for determining the number of chunks for the given number of *cores*.

Param exec [in] The executor object which will be used for scheduling of the loop iterations.

Param cores [in] The number of cores the number of chunks should be determined for.

Param num_tasks [in] The number of tasks the chunk size should be determined for

Private Functions

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*maximal_number_of_chunks_t*, *Parameters* &&params, *Executor* &&exec, *std*::size_t cores, *std*::size_t num_tasks)

struct **measure_iteration_t** : public *hpx*::*functional*::detail::tag_priority<*measure_iteration_t*>

#include <execution_parameters_fwd.hpp> Return the measured execution time for one iteration based on running the given function.

Note: The parameter *f* is expected to be a nullary function returning a `std::size_t` representing the number of iteration the function has already executed (i.e. which don't have to be scheduled anymore).

Param params [in] The executor parameters object to use for determining the chunk size for the given number of tasks *num_tasks*.
Param exec [in] The executor object which will be used for scheduling of the loop iterations.
Param f [in] The function which will be optionally scheduled using the given executor.
Param num_tasks [in] The number of tasks the chunk size should be determined for
Return The execution time for one of the tasks.

Private Functions

template<typename **Parameters**, typename **Executor**, typename **F**>
inline decltype(auto) friend **tag_fallback_invoke**(*measure_iteration_t*, *Parameters* &&params, *Executor* &&exec, *F* &&f, *std*::size_t num_tasks)

struct **null_parameters_t**

struct **processing_units_count_t** : public *hpx*::*functional*::detail::tag_priority<*processing_units_count_t*>

#include <execution_parameters_fwd.hpp> Retrieve the number of (kernel-)threads used by the associated executor.

Note: This calls params.processing_units_count(Executor&&) if it exists; otherwise it forwards the request to the executor parameters object.

Param params [in] The executor parameters object to use as a fallback if the executor does not expose
Param iteration_duration [in] The time one of the tasks require to be executed.
Param num_tasks [in] The number of tasks the number of cores should be determined for
Return The number of cores to use

Private Functions

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*processing_units_count_t*, *Parameters* &&params, *Executor* &&exec, *hpx*::*chrono*::steady_duration const &iteration_duration, *std*::size_t num_tasks)

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*processing_units_count_t* tag, *Parameters* &&params, *Executor* &&exec, *std*::size_t num_tasks = 0)

template<typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*processing_units_count_t*, *Executor* &&exec, *hpx*::*chrono*::steady_duration const &iteration_duration, *std*::size_t num_tasks)

template<typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*processing_units_count_t* tag, *Executor* &&exec, *std*::size_t num_tasks = 0)

struct **reset_thread_distribution_t** : public *hpx*::*functional*::detail::tag_priority<*reset_thread_distribution_t*>

#include <execution_parameters_fwd.hpp> Reset the internal round robin thread distribution scheme for the given executor.

Note: This calls params.reset_thread_distribution(exec) if it exists; otherwise it does nothing.

Param params [in] The executor parameters object to use for resetting the thread distribution scheme.

Param exec [in] The executor object to use.

Private Functions

template<typename **Parameters**, typename **Executor**>
inline decltype(auto) friend **tag_fallback_invoke**(*reset_thread_distribution_t*, *Parameters* &&params, *Executor* &&exec)

struct **with_processing_units_count_t** : public *hpx*::*functional*::detail::tag_priority<*with_processing_units_count_t*>

#include <execution_parameters_fwd.hpp> Generate a policy that supports setting the number of cores for execution.

hpx::execution::experimental::guided_chunk_size

Defined in header hpx/execution.hpp[694].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

struct **guided_chunk_size**

#include <guided_chunk_size.hpp> Iterations are dynamically assigned to threads in blocks as threads request those until no blocks remain to be assigned. Similar to dynamic_chunk_size except that the block size decreases each time a number of loop iterations is given to a thread. The size of the initial block is proportional to *number_of_iterations* / *number_of_cores*. Subsequent blocks are proportional to *number_of_iterations_remaining* / *number_of_cores*. The optional chunk size parameter defines the minimum block size. The default chunk size is 1.

Note: This executor parameters type is equivalent to OpenMP's GUIDED scheduling directive.

Public Functions

guided_chunk_size() = default

Construct an dynamic_chunk_size executor parameters object

Note: Default constructed dynamic_chunk_size executor parameter types will use a chunk size of '1'.

inline explicit constexpr **guided_chunk_size**(*std*::size_t min_chunk_size) noexcept

Construct a guided_chunk_size executor parameters object

Parameters **min_chunk_size** – [in] The optional minimal chunk size to use as the minimal number of loop iterations to schedule together. The default minimal chunk size is 1.

hpx::execution::experimental::num_cores

Defined in header hpx/execution.hpp[695].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[694] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp

[695] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp

namespace **execution**

namespace **experimental**

struct **num_cores**

#include <num_cores.hpp> Control number of cores in executors which need a functionality for setting the number of cores to be used by an algorithm directly

Public Functions

inline explicit constexpr **num_cores**(*std*::size_t cores = 1) noexcept

Construct a num_cores executor parameters object

Note: make sure the minimal number of cores is and the maximum number of cores is what's available to HPX

hpx::execution::experimental::persistent_auto_chunk_size

Defined in header hpx/execution.hpp[696].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

struct **persistent_auto_chunk_size**

#include <persistent_auto_chunk_size.hpp> Loop iterations are divided into pieces and then assigned to threads. The number of loop iterations combined is determined based on measurements of how long the execution of 1% of the overall number of iterations takes. This executor parameters type makes sure that as many loop iterations are combined as necessary to run for the amount of time specified.

[696] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp

Public Functions

inline explicit constexpr **persistent_auto_chunk_size**(*std*::uint64_t num_iters_for_timing = 0) noexcept

Construct an persistent_auto_chunk_size executor parameters object

Note: Default constructed persistent_auto_chunk_size executor parameter types will use 0 microseconds as the execution time for each chunk and 80 microseconds as the minimal time for which any of the scheduled chunks should run.

inline explicit **persistent_auto_chunk_size**(*hpx*::*chrono*::steady_duration const &time_cs, *std*::uint64_t num_iters_for_timing = 0) noexcept

Construct an persistent_auto_chunk_size executor parameters object

Parameters

- **time_cs** – The execution time for each chunk.
- **num_iters_for_timing** – [in] The number of iterations to use for measuring the execution time of one iteration

inline **persistent_auto_chunk_size**(*hpx*::*chrono*::steady_duration const &time_cs, *hpx*::*chrono*::steady_duration const &rel_time, *std*::uint64_t num_iters_for_timing = 0) noexcept

Construct an persistent_auto_chunk_size executor parameters object

Parameters

- **rel_time** – [in] The time duration to use as the minimum to decide how many loop iterations should be combined.
- **time_cs** – The execution time for each chunk.
- **num_iters_for_timing** – [in] The number of iterations to use for measuring the execution time of one iteration

hpx/execution/executors/polymorphic_executor.hpp

Defined in header hpx/execution/executors/polymorphic_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

template<typename **Sig**>

class **polymorphic_executor**

template<typename **R**, typename ...**Ts**>

class **polymorphic_executor**<*R*(*Ts*...)> : private *hpx*::*parallel*::*execution*::detail::polymorphic_executor_base

Public Types

template<typename>

using **future_type** = *hpx*::*future*<*R*>

Public Functions

inline constexpr **polymorphic_executor**() noexcept

inline **polymorphic_executor**(*polymorphic_executor* const &other)

inline **polymorphic_executor**(*polymorphic_executor* &&other) noexcept

inline *polymorphic_executor* &**operator=**(*polymorphic_executor* const &other)

inline *polymorphic_executor* &**operator=**(*polymorphic_executor* &&other) noexcept

template<typename **Exec**, typename **PE** = *std*::decay_t<*Exec*>, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*PE*, *polymorphic_executor*>>>
inline **polymorphic_executor**(*Exec* &&exec)

template<typename **Exec**, typename **PE** = *std*::decay_t<*Exec*>, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*PE*, *polymorphic_executor*>>>
inline *polymorphic_executor* &**operator=**(*Exec* &&exec)

inline void **reset**() noexcept

Private Types

using **base_type** = detail::polymorphic_executor_base

using **vtable** = detail::polymorphic_executor_vtable<*R*(*Ts*...)>

Private Functions

inline void **assign**(*std*::nullptr_t) noexcept

template<typename **Exec**>
inline void **assign**(*Exec* &&exec)

Private Static Functions

static inline constexpr *vtable* const ***get_empty_vtable**() noexcept

template<typename **T**>
static inline constexpr *vtable* const ***get_vtable**() noexcept

Friends

template<typename **F**>
inline friend void **tag_invoke**(*hpx*::*parallel*::*execution*::*post_t*, *polymorphic_executor* const &exec, *F* &&f, *Ts*... ts)

template<typename **F**>
inline friend *R* **tag_invoke**(*hpx*::*parallel*::*execution*::*sync_execute_t*, *polymorphic_executor* const &exec, *F* &&f, *Ts*... ts)

template<typename **F**>
inline friend *hpx*::*future*<*R*> **tag_invoke**(*hpx*::*parallel*::*execution*::*async_execute_t*, *polymorphic_executor* const &exec, *F* &&f, *Ts*... ts)

template<typename **F**, typename **Future**>
inline friend *hpx*::*future*<*R*> **tag_invoke**(*hpx*::*parallel*::*execution*::*then_execute_t*, *polymorphic_executor* const &exec, *F* &&f, *Future* &&predecessor, *Ts*&&... ts)

template<typename **F**, typename **Shape**>
inline friend *std*::vector<*R*> **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_sync_execute_t*, *polymorphic_executor* const &exec, *F* &&f, *Shape* const &s, *Ts*&&... ts)

template<typename **F**, typename **Shape**>
inline friend *std*::vector<*hpx*::*future*<*R*>> **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_async_execute_t*, *polymorphic_executor* const &exec, *F* &&f, *Shape* const &s, *Ts*&&... ts)

template<typename **F**, typename **Shape**>
inline friend *hpx*::*future*<*std*::vector<*R*>> **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_then_execute_t*, *polymorphic_executor* const &exec, *F* &&f, *Shape* const &s, *hpx*::*shared_future*<void> const &predecessor, *Ts*&&... ts)

hpx/execution/executors/rebind_executor.hpp

Defined in header hpx/execution/executors/rebind_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Typedefs

template<typename **ExPolicy**, typename **Executor**, typename **Parameters**>

using **rebind_executor_t** = typename *rebind_executor*<*ExPolicy*, *Executor*, *Parameters*>::type

Variables

constexpr struct *hpx*::*parallel*::*execution*::*create_rebound_policy_t* **create_rebound_policy**

struct **create_rebound_policy_t**

Public Functions

template<typename **ExPolicy**, typename **Executor**, typename **Parameters**>
inline constexpr decltype(auto) **operator()**(*ExPolicy*&&, *Executor* &&exec, *Parameters* &&parameters) const

template<typename **ExPolicy**, typename **Executor**, typename **Parameters**>

struct **rebind_executor**

#include <rebind_executor.hpp> Rebind the type of executor used by an execution policy. The execution category of Executor shall not be weaker than that of ExecutionPolicy.

Public Types

using **type** = typename policy_type::template rebind<executor_type, parameters_type>::type

The type of the rebound execution policy.

hpx::execution::experimental::static_chunk_size

Defined in header hpx/execution.hpp[697].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

[697] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp

struct **static_chunk_size**

#include <static_chunk_size.hpp> Loop iterations are divided into pieces of size *chunk_size* and then assigned to threads. If *chunk_size* is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

Note: This executor parameters type is equivalent to OpenMP's STATIC scheduling directive.

Public Functions

static_chunk_size() = default

Construct a static_chunk_size executor parameters object

Note: By default the number of loop iterations is determined from the number of available cores and the overall number of loop iterations to schedule.

inline explicit constexpr **static_chunk_size**(*std*::size_t chunk_size) noexcept

Construct a static_chunk_size executor parameters object

Parameters **chunk_size** – [in] The optional chunk size to use as the number of loop iterations to run on a single thread.

hpx/execution/traits/is_execution_policy.hpp

Defined in header hpx/execution/traits/is_execution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Variables

template<typename **T**>

constexpr bool **is_execution_policy_v** = *is_execution_policy*<*T*>::value

template<typename **T**>

constexpr bool **is_parallel_execution_policy_v** = *is_parallel_execution_policy*<*T*>::value

template<typename **T**>

constexpr bool **is_sequenced_execution_policy_v** = *is_sequenced_execution_policy*<*T*>::value

template<typename **T**>

constexpr bool **is_async_execution_policy_v** = *is_async_execution_policy*<*T*>::value

template<typename **T**>

struct **is_async_execution_policy** : public *hpx*::detail::is_async_execution_policy<*std*::decay_t<*T*>>

#include <is_execution_policy.hpp> Extension: Detect whether given execution policy makes algorithms asynchronous

i. The type *is_async_execution_policy* can be used to detect asynchronous execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

ii. If T is the type of a standard or implementation-defined execution policy, is_async_execution_policy<T> shall be publicly derived from integral_constant<bool, true>, otherwise from integral_constant<bool, false>.

iii. The behavior of a program that adds specializations for *is_async_execution_policy* is undefined.

template<typename **T**>

struct **is_execution_policy** : public *hpx*::detail::is_execution_policy<*std*::decay_t<*T*>>

#include <is_execution_policy.hpp>

i. The type *is_execution_policy* can be used to detect execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

ii. If T is the type of a standard or implementation-defined execution policy, is_execution_policy<T> shall be publicly derived from integral_constant<bool, true>, otherwise from integral_constant<bool, false>.

iii. The behavior of a program that adds specializations for *is_execution_policy* is undefined.

template<typename **T**>

struct **is_parallel_execution_policy** : public *hpx*::detail::is_parallel_execution_policy<*std*::decay_t<*T*>>

#include <is_execution_policy.hpp> Extension: Detect whether given execution policy enables parallelization

i. The type *is_parallel_execution_policy* can be used to detect parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

ii. If T is the type of a standard or implementation-defined execution policy, is_parallel_execution_policy<T> shall be publicly derived from integral_constant<bool, true>, otherwise from integral_constant<bool, false>.

iii. The behavior of a program that adds specializations for *is_parallel_execution_policy* is undefined.

template<typename **T**>

struct **is_sequenced_execution_policy** : public *hpx*::detail::is_sequenced_execution_policy<*std*::decay_t<*T*>>

#include <is_execution_policy.hpp> Extension: Detect whether given execution policy does not enable parallelization

i. The type *is_sequenced_execution_policy* can be used to detect non-parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

ii. If T is the type of a standard or implementation-defined execution policy, is_sequenced_execution_policy<T> shall be publicly derived from integral_constant<bool, true>, otherwise from integral_constant<bool, false>.

iii. The behavior of a program that adds specializations for *is_sequenced_execution_policy* is undefined.

execution_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/execution_base/execution.hpp

Defined in header hpx/execution_base/execution.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

struct **parallel_execution_tag**

#include <execution.hpp> Function invocations executed by a group of parallel execution agents execute in unordered fashion. Any such invocations executing in the same thread are indeterminately sequenced with respect to each other.

Note: parallel_execution_tag is weaker than sequenced_execution_tag.

struct **sequenced_execution_tag**

#include <execution.hpp> Function invocations executed by a group of sequential execution agents execute in sequential order.

struct **unsequenced_execution_tag**

#include <execution.hpp> Function invocations executed by a group of vector execution agents are permitted to execute in unordered fashion when executed in different threads, and un-sequenced with respect to one another when executed in the same thread.

Note: unsequenced_execution_tag is weaker than parallel_execution_tag.

namespace **parallel**

namespace **execution**

Variables

hpx::*parallel*::*execution*::*sync_execute_t* **sync_execute**

hpx::*parallel*::*execution*::*async_execute_t* **async_execute**

hpx::*parallel*::*execution*::*then_execute_t* **then_execute**

hpx::*parallel*::*execution*::*post_t* **post**

hpx::*parallel*::*execution*::*bulk_sync_execute_t* **bulk_sync_execute**

hpx::*parallel*::*execution*::*bulk_async_execute_t* **bulk_async_execute**

hpx::*parallel*::*execution*::*bulk_then_execute_t* **bulk_then_execute**

hpx::*parallel*::*execution*::*async_invoke_t* **async_invoke**

hpx::*parallel*::*execution*::*sync_invoke_t* **sync_invoke**

struct **async_execute_t** : public *hpx*::*functional*::detail::tag_fallback<*async_execute_t*>

#include <execution.hpp> Customization point for asynchronous execution agent creation.

This asynchronously creates a single function invocation f() using the associated executor.

Note: Executors have to implement only `async_execute()`. All other functions will be emulated by this or other customization points in terms of this single basic primitive. However, some executors will naturally specialize all operations for maximum efficiency.

Note: This is valid for one way executors (calls make_ready_future(exec.sync_execute(f, ts…) if it exists) and for two way executors (calls exec.async_execute(f, ts…) if it exists).

Param exec [in] The executor object to use for scheduling of the function *f*.
Param f [in] The function which will be scheduled using the given executor.
Param ts [in] Additional arguments to use to invoke *f*.
Return f(ts…)'s result through a future

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*async_execute_t*, *Executor* &&exec, *F* &&f, *Ts*&&... ts)

struct **async_invoke_t** : public *hpx*::*functional*::detail::tag_fallback<*async_invoke_t*>

#include <execution.hpp> Asynchronously invoke the given set of nullary functions, each on its own execution agent

This creates a group of function invocations whose ordering is given by the execution_category associated with the executor.

All exceptions thrown by invocations of the functions are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This calls exec.async_invoke(fs…) if it exists; otherwise it executes async_execute(fs) for each fs.

Param exec [in] The executor object to use for scheduling of the functions *fs*.
Param fs [in] The functions which will be scheduled using the given executor.
Return The return type of *executor_type::async_invoke* if defined by *executor_type*. Otherwise a future<void> representing finishing the execution of all functions *fs*.

Private Functions

template<typename **Executor**, typename **F**, typename ...**Fs**>
inline decltype(auto) friend **tag_fallback_invoke**(*async_invoke_t*, *Executor* &&exec, *F* &&f, *Fs*&&... fs)

struct **bulk_async_execute_t** : public *hpx*::*functional*::detail::tag_fallback<*bulk_async_execute_t*>

#include <execution.hpp> Bulk form of asynchronous execution agent creation.

This asynchronously creates a group of function invocations f(i) whose ordering is given by the execution_category associated with the executor.

Here *i* takes on all values in the index space implied by shape. All exceptions thrown by invocations of f(i) are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This is deliberately different from the bulk_async_execute customization points specified in P0443.The bulk_async_execute customization point defined here is more generic and is used as the workhorse for implementing the specified APIs.

Note: This calls exec.bulk_async_execute(f, shape, ts…) if it exists; otherwise it executes async_execute(f, shape, ts…) as often as needed.

Param exec [in] The executor object to use for scheduling of the function *f*.

Param f [in] The function which will be scheduled using the given executor.

Param shape [in] The shape objects which defines the iteration boundaries for the arguments to be passed to *f*.

Param ts [in] Additional arguments to use to invoke *f*.

Return The return type of *executor_type::bulk_async_execute* if defined by *executor_type*. Otherwise a vector of futures holding the returned values of each invocation of *f*.

Private Functions

template<typename **Executor**, typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*bulk_async_execute_t*, *Executor* &&exec, *F* &&f, *Shape* const &shape, *Ts*&&... ts)

template<typename **Executor**, typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*bulk_async_execute_t* tag, *Executor* &&exec, *F* &&f, *Shape* const &shape, *Ts*&&... ts)

struct **bulk_sync_execute_t** : public *hpx*::*functional*::detail::tag_fallback<*bulk_sync_execute_t*>

#include <execution.hpp> Bulk form of synchronous execution agent creation.

This synchronously creates a group of function invocations f(i) whose ordering is given by the execution_category associated with the executor. The function synchronizes the execution of all scheduled functions with the caller.

Here *i* takes on all values in the index space implied by shape. All exceptions thrown by invocations of f(i) are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This is deliberately different from the bulk_sync_execute customization points specified in P0443.The bulk_sync_execute customization point defined here is more generic and is used as the workhorse for implementing the specified APIs.

Note: This calls exec.bulk_sync_execute(f, shape, ts…) if it exists; otherwise it executes sync_execute(f, shape, ts…) as often as needed.

Param exec [in] The executor object to use for scheduling of the function *f*.

Param f [in] The function which will be scheduled using the given executor.

Param shape [in] The shape objects which defines the iteration boundaries for the arguments to be passed to *f*.

Param ts [in] Additional arguments to use to invoke *f*.

Return The return type of *executor_type::bulk_sync_execute* if defined by *executor_type*. Otherwise a vector holding the returned values of each invocation of *f* except when *f* returns void, which case void is returned.

Private Functions

template<typename **Executor**, typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*bulk_sync_execute_t*, *Executor* &&exec, *F* &&f, *Shape* const &shape, *Ts*&&... ts)

template<typename **Executor**, typename **F**, typename **Shape**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*bulk_sync_execute_t* tag, *Executor* &&exec, *F* &&f, *Shape* const &shape, *Ts*&&... ts)

struct **bulk_then_execute_t** : public *hpx*::*functional*::detail::tag_fallback<*bulk_then_execute_t*>

#include <execution.hpp> Bulk form of execution agent creation depending on a given future.

This creates a group of function invocations f(i) whose ordering is given by the execution_category associated with the executor.

Here *i* takes on all values in the index space implied by shape. All exceptions thrown by invocations of f(i) are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This is deliberately different from the then_sync_execute customization points specified in P0443.The bulk_then_execute customization point defined here is more generic and is used as the workhorse for implementing the specified APIs.

Note: This calls exec.bulk_then_execute(f, shape, pred, ts...) if it exists; otherwise it executes sync_execute(f, shape, pred.share(), ts...) (if this executor is also an OneWayExecutor), or async_execute(f, shape, pred.share(), ts...) (if this executor is also a TwoWayExecutor) - as often as needed.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param f [in] The function which will be scheduled using the given executor.
Param shape [in] The shape objects which defines the iteration boundaries for the arguments to be passed to *f*.
Param predecessor [in] The future object the execution of the given function depends on.
Param ts [in] Additional arguments to use to invoke *f*.
Return The return type of *executor_type::bulk_then_execute* if defined by *executor_type*. Otherwise a vector holding the returned values of each invocation of *f*.

Private Functions

template<typename **Executor**, typename **F**, typename **Shape**, typename **Future**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*bulk_then_execute_t*, *Executor* &&exec, *F* &&f, *Shape* const &shape, *Future* &&predecessor, *Ts*&&... ts)

template<typename **Executor**, typename **F**, typename **Shape**, typename **Future**, typename ...**Ts**>

inline decltype(auto) friend **tag_fallback_invoke**(*bulk_then_execute_t* tag, *Executor* &&exec, *F* &&f, *Shape* const &shape, *Future* &&predecessor, *Ts*&&... ts)

struct **post_t** : public *hpx*::*functional*::detail::tag_fallback<*post_t*>

#include <execution.hpp> Customization point for asynchronous fire & forget execution agent creation.

This asynchronously (fire & forget) creates a single function invocation f() using the associated executor.

Note: This is valid for two way executors (calls exec.post(f, ts…), if available, otherwise it calls exec.async_execute(f, ts…) while discarding the returned future), and for non-blocking two way executors (calls exec.post(f, ts…) if it exists).

Param exec [in] The executor object to use for scheduling of the function *f*.
Param f [in] The function which will be scheduled using the given executor.
Param ts [in] Additional arguments to use to invoke *f*.

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*post_t*, *Executor* &&exec, *F* &&f, *Ts*&&... ts)

struct **sync_execute_t** : public *hpx*::*functional*::detail::tag_fallback<*sync_execute_t*>

#include <execution.hpp> Customization point for synchronous execution agent creation.

This synchronously creates a single function invocation f() using the associated executor. The execution of the supplied function synchronizes with the caller

---

Note: It will call tag_invoke(sync_execute_t, exec, f, ts…) if it exists. For two-way executors it will invoke asynch_execute_t and wait for the task's completion before returning.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param f [in] The function which will be scheduled using the given executor.
Param ts [in] Additional arguments to use to invoke *f*.
Return f(ts…)'s result

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*sync_execute_t*, *Executor* &&exec, *F* &&f, *Ts*&&... ts)

struct **sync_invoke_t** : public *hpx*::*functional*::detail::tag_fallback<*sync_invoke_t*>

#include <execution.hpp> Synchronously invoke the given set of nullary functions, each on its own execution agent

This creates a group of function invocations whose ordering is given by the execution_category associated with the executor.

All exceptions thrown by invocations of the functions are reported in a manner consistent with parallel algorithm execution through the returned future.

Note: This calls exec.sync_invoke(fs…) if it exists; otherwise it executes sync_execute(fs) for each fs.

Param exec [in] The executor object to use for scheduling of the functions *fs*.
Param fs [in] The functions which will be scheduled using the given executor.
Return The return type of *executor_type::async_invoke* if defined by *executor_type*.

Private Functions

template<typename **Executor**, typename **F**, typename ...**Fs**>
inline decltype(auto) friend **tag_fallback_invoke**(*sync_invoke_t*, *Executor* &&exec, *F* &&f, *Fs*&&... fs)

struct **then_execute_t** : public *hpx*::*functional*::detail::tag_fallback<*then_execute_t*>

#include <execution.hpp> Customization point for execution agent creation depending on a given future.

This creates a single function invocation f() using the associated executor after the given future object has become ready.

Note: This is valid for two way executors (calls exec.then_execute(f, predecessor, ts…) if it exists) and for one way executors (calls predecessor.then(bind(f, ts…))).

Param exec [in] The executor object to use for scheduling of the function *f*.
Param f [in] The function which will be scheduled using the given executor.
Param predecessor [in] The future object the execution of the given function depends on.
Param ts [in] Additional arguments to use to invoke *f*.
Return f(ts…)'s result through a future

Private Functions

template<typename **Executor**, typename **F**, typename **Future**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*then_execute_t*, *Executor* &&exec, *F* &&f, *Future* &&predecessor, *Ts*&&... ts)

hpx/execution_base/receiver.hpp

Defined in header hpx/execution_base/receiver.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Functions

template<typename **R**, typename ...**As**>
void **set_value**(*R* &&r, *As*&&... as)

set_value is a customization point object. The expression `hpx::execution::set_value(r, as...)` is equivalent to:

- `r.set_value(as...)`, if that expression is valid. If the function selected does not send the value(s) `as...` to the Receiver `r`'s value channel, the program is ill-formed (no diagnostic required).
- Otherwise, `set_value(r, as…), if that expression is valid, with overload resolution performed in a context that include the declaration `void set_value();`
- Otherwise, the expression is ill-formed.

The customization is implemented in terms of `hpx::functional::tag_invoke`.

template<typename **R**>
void **set_stopped**(*R* &&r)

set_stopped is a customization point object. The expression `hpx::execution::set_stopped(r)` is equivalent to:

- `r.set_stopped()`, if that expression is valid. If the function selected does not signal the Receiver `r`'s done channel, the program is ill-formed (no diagnostic required).
- Otherwise, `set_stopped(r), if that expression is valid, with overload resolution performed in a context that include the declaration `void set_stopped();`
- Otherwise, the expression is ill-formed.

The customization is implemented in terms of `hpx::functional::tag_invoke`.

template<typename **R**, typename **E**>
void **set_error**(*R* &&r, *E* &&e)

set_error is a customization point object. The expression `hpx::execution::set_error(r, e)` is equivalent to:

- `r.set_stopped(e)`, if that expression is valid. If the function selected does not send the error `e` the Receiver `r`'s error channel, the program is ill-formed (no diagnostic required).
- Otherwise, `set_error(r, e), if that expression is valid, with overload resolution performed in a context that include the declaration `void set_error();`
- Otherwise, the expression is ill-formed.

The customization is implemented in terms of `hpx::functional::tag_invoke`.

Variables

hpx::*execution*::*experimental*::*set_value_t* **set_value**

hpx::*execution*::*experimental*::*set_error_t* **set_error**

hpx::*execution*::*experimental*::*set_stopped_t* **set_stopped**

template<typename **T**, typename **E** = *std*::exception_ptr>

constexpr bool **is_receiver_v** = *is_receiver*<*T*, *E*>::value

template<typename **T**, typename **CS**>

constexpr bool **is_receiver_of_v** = *is_receiver_of*<*T*, *CS*>::value

template<typename **T**, typename **CS**>

constexpr bool **is_nothrow_receiver_of_v** = *is_nothrow_receiver_of*<*T*, *CS*>::value

template<typename **T**, typename **CS**>

struct **is_nothrow_receiver_of** : public *hpx*::*execution*::*experimental*::detail::is_nothrow_receiver_of_impl<*is_receiver_v*<*T*> && *is_receiver_of_v*<*T*, *CS*>, *T*, *CS*>

template<typename **T**, typename **E**>

struct **is_receiver**

#include <receiver.hpp> Receiving values from asynchronous computations is handled by the `Receiver` concept. A `Receiver` needs to be able to receive an error or be marked as being canceled. As such, the Receiver concept is defined by having the following two customization points defined, which form the completion-signal operations:

- `hpx::execution::experimental::set_stopped` * `hpx::execution::experimental::set_error`

Those two functions denote the completion-signal operations. The Receiver contract is as follows:

- None of a Receiver's completion-signal operation shall be invoked before `hpx::execution::experimental::start` has been called on the operation state object that was returned by connecting a Receiver to a sender `hpx::execution::experimental::connect`.
- Once `hpx::execution::start` has been called on the operation state object, exactly one of the Receiver's completion-signal operation shall complete without an exception before the Receiver is destroyed

Once one of the Receiver's completion-signal operation has been completed without throwing an exception, the Receiver contract has been satisfied. In other words: The asynchronous operation has been completed.

See also:

hpx::execution::experimental::is_receiver_of

template<typename **T**, typename **CS**>

struct **is_receiver_of**

#include <receiver.hpp> The `receiver_of` concept is a refinement of the `Receiver` concept by requiring one additional completion-signal operation:

- `hpx::execution::set_value`

The `receiver_of` concept takes a receiver and an instance of the `completion_signatures<>` class template. The `receiver_of` concept, rather than accepting a receiver and some value types, is changed to take a receiver and an instance of the `completion_signatures<>` class template. A sender uses `completion_signatures<>` to describe the signals with which it completes. The `receiver_of` concept ensures that a particular receiver is capable of receiving those signals.

This completion-signal operation adds the following to the Receiver's contract:

- If `hpx::execution::set_value` exits with an exception, it is still valid to call `hpx::execution::set_error` or `hpx::execution::set_stopped`

See also:

hpx::execution::traits::is_receiver

struct **set_error_t** : public *hpx*::*functional*::tag_noexcept<*set_error_t*>

struct **set_stopped_t** : public *hpx*::*functional*::tag_noexcept<*set_stopped_t*>

struct **set_value_t** : public *hpx*::*functional*::tag<*set_value_t*>

hpx/execution_base/traits/is_executor_parameters.hpp

Defined in header hpx/execution_base/traits/is_executor_parameters.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **Executor**>

struct *hpx*::*parallel*::*execution*::**extract_executor_parameters**<*Executor*, *std*::void_t<typename *Executor*::executor_parameters_type>>

Public Types

using **type** = typename *Executor*::executor_parameters_type

template<typename **Parameters**>

struct **extract_has_variable_chunk_size**<*Parameters*, *std*::void_t<typename *Parameters*::has_variable_chunk_size>> : public true_type

template<typename **Parameters**>

struct **extract_has_variable_chunk_size**<::*std*::reference_wrapper<*Parameters*>> : public *hpx*::*parallel*::*execution*::*extract_has_variable_chunk_size*<*Parameters*>

template<typename **Parameters**>

struct **extract_invokes_testing_function**<::*std*::reference_wrapper<*Parameters*>> : public *hpx*::*parallel*::*execution*::*extract_invokes_testing_function*<*Parameters*>

namespace **hpx**

namespace **parallel**

namespace **execution**

Typedefs

template<typename **Executor**>

using **extract_executor_parameters_t** = typename *extract_executor_parameters*<*Executor*>::type

Variables

template<typename **Parameters**>

constexpr bool **extract_has_variable_chunk_size_v** = *extract_has_variable_chunk_size*<*Parameters*>::value

template<typename **Parameters**>

constexpr bool **extract_invokes_testing_function_v** = *extract_invokes_testing_function*<*Parameters*>::value

template<typename **T**>

constexpr bool **is_executor_parameters_v** = *is_executor_parameters*<*T*>::value

template<typename **Executor**, typename **Enable** = void>

struct **extract_executor_parameters**

Public Types

using **type** = *sequential_executor_parameters*

template<typename Executor> executor_parameters_type > >

Public Types

using **type** = typename Executor::executor_parameters_type

template<typename **Parameters**, typename **Enable** = void>

struct **extract_has_variable_chunk_size** : public false_type

template<typename Parameters> has_variable_chunk_size > > : public true_type

template<typename Parameters> reference_wrapper< Parameters > > : public hpx::parallel::exe

template<typename **Parameters**, typename **Enable** = void>

struct **extract_invokes_testing_function** : public false_type

template<typename Parameters> reference_wrapper< Parameters > > : public hpx::parallel::exe

template<typename **T**>

struct **is_executor_parameters** : public detail::is_executor_parameters<*std*::decay_t<*T*>>

struct **sequential_executor_parameters**

namespace **traits**

Variables

template<typename **T**>

constexpr bool **is_executor_parameters_v** = *is_executor_parameters*<*T*>::value

template<typename **Parameters**, typename **Enable**>

struct **is_executor_parameters**

executors

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/executors/annotating_executor.hpp

Defined in header hpx/executors/annotating_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Functions

template<typename **Tag**, typename **BaseExecutor**, typename **Property**>
auto **tag_invoke**(*Tag* tag, *annotating_executor*<*BaseExecutor*> const &exec, *Property* &&prop) -> decltype(*annotating_executor*<*BaseExecutor*>(*std*::declval<*Tag*>()(*std*::declval<*BaseExecutor*>(), *std*::declval<*Property*>())))

template<typename **Tag**, typename **BaseExecutor**>
auto **tag_invoke**(*Tag* tag, *annotating_executor*<*BaseExecutor*> const &exec) -> decltype(*std*::declval<*Tag*>()(*std*::declval<*BaseExecutor*>()))

template<typename **Executor**>
constexpr auto **tag_fallback_invoke**(with_annotation_t, *Executor* &&exec, char const *annotation)

template<typename **Executor**>
auto **tag_fallback_invoke**(with_annotation_t, *Executor* &&exec, *std*::string annotation)

template<typename **BaseExecutor**>

struct **annotating_executor**

#include <annotating_executor.hpp> A annotating_executor wraps any other executor and adds the capability to add annotations to the launched threads.

Public Functions

template<typename **Executor**, typename **Enable** = *std*::enable_if_t<*hpx*::*traits*::is_executor_any_v<*Executor*> && !*std*::is_same_v<*std*::decay_t<*Executor*>, *annotating_executor*>>>
inline explicit constexpr **annotating_executor**(*Executor* &&exec, char const *annotation = nullptr)

template<typename **Executor**, typename **Enable** = *std*::enable_if_t<*hpx*::*traits*::is_executor_any_v<*Executor*>>>
inline explicit **annotating_executor**(*Executor* &&exec, *std*::string annotation)

namespace **parallel**

namespace **execution**

hpx/executors/current_executor.hpp

Defined in header hpx/executors/current_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Typedefs

typedef *hpx*::*execution*::*parallel_executor* **instead**

namespace **this_thread**

Functions

hpx::*execution*::*parallel_executor* **get_executor**(*error_code* &ec = *throws*)

Returns a reference to the executor that was used to create the current thread.

Throws `If – &ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

namespace **threads**

Functions

hpx::*execution*::*parallel_executor* **get_executor**(thread_id_type const &id, *error_code* &ec = *throws*)

Returns a reference to the executor that was used to create the given thread.

Throws `If – &ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

hpx/executors/exception_list.hpp

Defined in header hpx/executors/exception_list.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

hpx::execution::seq, hpx::execution::par, hpx::execution::par_unseq, hpx::execution::task, hpx::execution::sequenced_policy, hpx::execution::parallel_policy, hpx::execution::parallel_unsequenced_policy, hpx::execution::sequenced_task_policy, hpx::execution::parallel_task_policy

Defined in header hpx/execution.hpp[698].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

Typedefs

using **sequenced_task_policy** = detail::sequenced_task_policy_shim<*sequenced_executor*, *hpx*::*traits*::executor_parameters_type_t<*sequenced_executor*>>

Extension: The class sequenced_task_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may not be parallelized (has to run sequentially).

The algorithm returns a future representing the result of the corresponding algorithm when invoked with the sequenced_policy.

using **sequenced_policy** = detail::sequenced_policy_shim<*sequenced_executor*, *hpx*::*traits*::executor_parameters_type_t<*sequenced_executor*>>

The class sequenced_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.

using **parallel_task_policy** = detail::parallel_task_policy_shim<*parallel_executor*, *hpx*::*traits*::executor_parameters_type_t<*parallel_executor*>>

Extension: The class parallel_task_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

The algorithm returns a future representing the result of the corresponding algorithm when invoked with the parallel_policy.

using **parallel_policy** = detail::parallel_policy_shim<*parallel_executor*, *hpx*::*traits*::executor_parameters_type_t<*parallel_executor*>>

The class parallel_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

using **parallel_unsequenced_task_policy** = detail::parallel_unsequenced_task_policy_shim<*parallel_executor*, *hpx*::*traits*::executor_parameters_type_t<*parallel_executor*>>

The class parallel_unsequenced_task_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized.

[698] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/execution.hpp

using **parallel_unsequenced_policy** = detail::parallel_unsequenced_policy_shim<*parallel_executor*, *hpx*::*traits*::executor_parameters_type_t<*parallel_executor*>>

The class parallel_unsequenced_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized.

using **unsequenced_task_policy** = detail::unsequenced_task_policy_shim<*sequenced_executor*, *hpx*::*traits*::executor_parameters_type_t<*sequenced_executor*>>

The class unsequenced_task_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized.

using **unsequenced_policy** = detail::unsequenced_policy_shim<*sequenced_executor*, *hpx*::*traits*::executor_parameters_type_t<*sequenced_executor*>>

The class unsequenced_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized.

Variables

constexpr *task_policy_tag* **task** = {}

constexpr *non_task_policy_tag* **non_task** = {}

constexpr *sequenced_policy* **seq** = {}

Default sequential execution policy object.

constexpr *parallel_policy* **par** = {}

Default parallel execution policy object.

constexpr *parallel_unsequenced_policy* **par_unseq** = {}

Default vector execution policy object.

constexpr *unsequenced_policy* **unseq** = {}

Default vector execution policy object.

struct **non_task_policy_tag** : public *hpx*::*execution*::*experimental*::*to_non_task_t*

struct **task_policy_tag** : public *hpx*::*execution*::*experimental*::*to_task_t*

namespace **experimental**

template<>

struct **is_execution_policy_mapping**<*non_task_policy_tag*> : public true_type

template<>

struct **is_execution_policy_mapping**<*task_policy_tag*> : public true_type

hpx/executors/execution_policy_annotation.hpp

Defined in header hpx/executors/execution_policy_annotation.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Functions

template<typename **ExPolicy**>
constexpr decltype(auto) **tag_invoke**(*hpx*::*execution*::*experimental*::with_annotation_t, *ExPolicy* &&policy, char const *annotation)

template<typename **ExPolicy**>
decltype(auto) **tag_invoke**(*hpx*::*execution*::*experimental*::with_annotation_t, *ExPolicy* &&policy, *std*::string annotation)

template<typename **ExPolicy**>
constexpr decltype(auto) **tag_invoke**(*hpx*::*execution*::*experimental*::get_annotation_t, *ExPolicy* &&policy)

hpx/executors/execution_policy_mappings.hpp

Defined in header hpx/executors/execution_policy_mappings.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Variables

template<typename **Tag**>

constexpr bool **is_execution_policy_mapping_v** = *is_execution_policy_mapping*<*Tag*>::value

hpx::*execution*::*experimental*::*to_non_par_t* **to_non_par**

hpx::*execution*::*experimental*::*to_par_t* **to_par** *hpx*::*execution*::*experimental*::*to_non_task_t* **to_non_task**

hpx::*execution*::*experimental*::*to_task_t* **to_task**

hpx::*execution*::*experimental*::*to_non_unseq_t* **to_non_unseq**

hpx::*execution*::*experimental*::*to_unseq_t* **to_unseq**

template<typename **Tag**>

struct **is_execution_policy_mapping** : public false_type

template<>

struct **is_execution_policy_mapping**<*to_non_par_t*> : public true_type

template<>

struct **is_execution_policy_mapping**<*to_non_task_t*> : public true_type

template<>

struct **is_execution_policy_mapping**<*to_non_unseq_t*> : public true_type

template<>

struct **is_execution_policy_mapping**<*to_par_t*> : public true_type

template<>

struct **is_execution_policy_mapping**<*to_task_t*> : public true_type

template<>

struct **is_execution_policy_mapping**<*to_unseq_t*> : public true_type

struct **to_non_par_t** : public *hpx*::*functional*::detail::tag_fallback<*to_non_par_t*>

Private Functions

template<typename **ExPolicy**>
inline constexpr decltype(auto) friend **tag_fallback_invoke**(*to_non_par_t*, *ExPolicy* &&policy) noexcept

struct **to_non_task_t** : public *hpx*::*functional*::detail::tag_fallback<*to_non_task_t*>

Subclassed by *hpx::execution::non_task_policy_tag*

Private Functions

template<typename **ExPolicy**>
inline constexpr decltype(auto) friend **tag_fallback_invoke**(*to_non_task_t*, *ExPolicy* &&policy) noexcept

struct **to_non_unseq_t** : public *hpx*::*functional*::detail::tag_fallback<*to_non_unseq_t*>

Private Functions

template<typename **ExPolicy**>
inline constexpr decltype(auto) friend **tag_fallback_invoke**(*to_non_unseq_t*, *ExPolicy* &&policy) noexcept

struct **to_par_t** : public *hpx*::*functional*::detail::tag_fallback<*to_par_t*>

Private Functions

template<typename **ExPolicy**>
inline constexpr decltype(auto) friend **tag_fallback_invoke**(*to_par_t*, *ExPolicy* &&policy) noexcept

struct **to_task_t** : public *hpx*::*functional*::detail::tag_fallback<*to_task_t*>

Subclassed by *hpx::execution::task_policy_tag*

Private Functions

template<typename **ExPolicy**>
inline constexpr decltype(auto) friend **tag_fallback_invoke**(*to_task_t*, *ExPolicy* &&policy) noexcept

struct **to_unseq_t** : public *hpx*::*functional*::detail::tag_fallback<*to_unseq_t*>

Private Functions

template<typename **ExPolicy**>
inline constexpr decltype(auto) friend **tag_fallback_invoke**(*to_unseq_t*, *ExPolicy* &&policy) noexcept

hpx/executors/execution_policy_parameters.hpp

Defined in header hpx/executors/execution_policy_parameters.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Functions

template<typename **ExPolicy**>
constexpr decltype(auto) **tag_invoke**(*with_processing_units_count_t*, *ExPolicy* &&policy, *std*::size_t num_cores)

template<typename **ExPolicy**, typename **Params**>
constexpr decltype(auto) **tag_invoke**(*with_processing_units_count_t*, *ExPolicy* &&policy, *Params* &&params)

template<typename **ParametersProperty**, typename **ExPolicy**, typename **Params**>
constexpr decltype(auto) **tag_fallback_invoke**(*ParametersProperty*, *ExPolicy* &&policy, *Params* &&params)

template<typename **ParametersProperty**, typename **ExPolicy**, typename ...**Ts**>
constexpr auto **tag_fallback_invoke**(*ParametersProperty* prop, *ExPolicy* &&policy, *Ts*&&... ts) -> decltype(*std*::declval<*ParametersProperty*>()(*std*::declval<typename *std*::decay_t<*ExPolicy*>::executor_type>(), *std*::declval<*Ts*>()...))

hpx/executors/execution_policy_scheduling_property.hpp

Defined in header hpx/executors/execution_policy_scheduling_property.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Functions

template<typename **Tag**, typename **ExPolicy**, typename **Property**>
constexpr decltype(auto) **tag_invoke**(*Tag* tag, *ExPolicy* &&policy, *Property* prop)

template<typename **Tag**, typename **ExPolicy**>
constexpr decltype(auto) **tag_invoke**(*Tag* tag, *ExPolicy* &&policy)

hpx/executors/explicit_scheduler_executor.hpp

Defined in header hpx/executors/explicit_scheduler_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Functions

template<typename **BaseScheduler**>
explicit **explicit_scheduler_executor**(*BaseScheduler* &&sched) -> *explicit_scheduler_executor*<*std*::decay_t<*BaseScheduler*>>

template<typename **Tag**, typename **BaseScheduler**, typename **Property**>
auto **tag_invoke**(*Tag* tag, *explicit_scheduler_executor*<*BaseScheduler*> const &exec, *Property* &&prop) -> decltype(*explicit_scheduler_executor*<*BaseScheduler*>(*std*::declval<*Tag*>()(*std*::declval<*BaseSchedul*
std::declval<*Property*>())))

template<typename **Tag**, typename **BaseScheduler**>
auto **tag_invoke**(*Tag* tag, *explicit_scheduler_executor*<*BaseScheduler*> const &exec) -> decltype(*std*::declval<*Tag*>()(*std*::declval<*BaseScheduler*>()))

template<typename **BaseScheduler**>

struct **explicit_scheduler_executor**

namespace **parallel**

namespace **execution**

hpx/executors/fork_join_executor.hpp

Defined in header hpx/executors/fork_join_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/executors/parallel_executor.hpp

Defined in header hpx/executors/parallel_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

Typedefs

using **parallel_executor** = *parallel_policy_executor*<*hpx*::*launch*>

Functions

template<typename **Tag**, typename **Policy**, typename **Property**>
auto **tag_invoke**(*Tag* tag, *parallel_policy_executor*<*Policy*> const &exec, *Property* &&prop) -> decltype(*std*::declval<*parallel_policy_executor*<*Policy*>>().policy(*std*::declval<*Tag*>()(*std*::declval<*Policy* *std*::declval<*Property*>())), *parallel_policy_executor*<*Policy*>())

template<typename **Tag**, typename **Policy**>
auto **tag_invoke**(*Tag* tag, *parallel_policy_executor*<*Policy*> const &exec) -> decltype(*std*::declval<*Tag*>()(*std*::declval<*Policy*>()))

template<typename **Policy**>

struct **parallel_policy_executor**

#include <parallel_executor.hpp> A *parallel_executor* creates groups of parallel execution agents which execute in threads implicitly created by the executor. This executor prefers continuing with the creating thread first before executing newly created threads.

This executor conforms to the concepts of a TwoWayExecutor, and a BulkTwoWayExecutor

Public Types

using **execution_category** = *std*::conditional_t<*std*::is_same_v<*Policy*, *launch*::sync_policy>, *sequenced_execution_tag*, *parallel_execution_tag*>

Associate the parallel_execution_tag executor tag type as a default with this executor, except if the given launch policy is synch.

using **executor_parameters_type** = *experimental*::*default_parameters*

Associate the default_parameters executor parameters type as a default with this executor.

Public Functions

inline explicit constexpr **parallel_policy_executor**(*threads*::*thread_priority* priority, *threads*::*thread_stacksize* stacksize = *threads*::*thread_stacksize*::*default_*, *threads*::*thread_schedule_hint* schedulehint = {}, *Policy* l = *parallel*::*execution*::detail::get_default_policy<*Policy*>::call(), *std*::size_t hierarchical_threshold = hierarchical_threshold_default_)

Create a new parallel executor.

inline explicit constexpr **parallel_policy_executor**(*threads*::*thread_stacksize* stacksize, *threads*::*thread_schedule_hint* schedulehint = {}, *Policy* l = *parallel*::*execution*::detail::get_default_policy<*Policy*>::call())

inline explicit constexpr **parallel_policy_executor**(*threads*::*thread_schedule_hint* schedulehint, *Policy* l = *parallel*::*execution*::detail::get_default_policy<*Policy*>::call())

inline explicit constexpr **parallel_policy_executor**(*Policy* l)

inline constexpr **parallel_policy_executor**()

inline explicit constexpr **parallel_policy_executor**(*threads*::*thread_pool_base* *pool, *Policy* l, *std*::size_t hierarchical_threshold = hierarchical_threshold_default_)

inline explicit constexpr **parallel_policy_executor**(*threads*::*thread_pool_base* *pool, *threads*::*thread_priority* priority = *threads*::*thread_priority*::*default_*, *threads*::*thread_stacksize* stacksize = *threads*::*thread_stacksize*::*default_*, *threads*::*thread_schedule_hint* schedulehint = {}, *Policy* l = *parallel*::*execution*::detail::get_default_policy<*Policy*>::call(), *std*::size_t hierarchical_threshold = hierarchical_threshold_default_)

inline constexpr void **set_hierarchical_threshold**(*std*::size_t threshold) noexcept

template<typename **Parameters**>
inline *std*::size_t **processing_units_count**(*Parameters*&&, *hpx*::*chrono*::steady_duration const& = *hpx*::*chrono*::null_duration, *std*::size_t = 0) const

Friends

**template<typename Executor_> inline friend constexpr friend auto tag_invoke (hpx::parallel:
Executor_ const &exec, std::size_t num_cores) noexcept**

**template<typename Parameters> inline friend constexpr friend std::size_t tag_invoke (hpx::p
Parameters &&, parallel_policy_executor const &exec,
hpx::chrono::steady_duration const &=hpx::chrono::null_duration,
std::size_t=0)**

**template<typename Executor_> inline friend constexpr friend auto tag_invoke (hpx::execution
Executor_ const &exec, std::size_t first_core) noexcept**

**inline friend constexpr friend std::size_t tag_invoke (hpx::execution::experimental::get_fi
parallel_policy_executor const &exec) noexcept**

inline friend auto **tag_invoke**(*hpx*::*execution*::*experimental*::get_processing_units_mask_t, *parallel_policy_executor* const &exec)

inline friend auto **tag_invoke**(*hpx*::*execution*::*experimental*::get_cores_mask_t, *parallel_policy_executor* const &exec)

namespace **parallel**

namespace **execution**

hpx/executors/parallel_executor_aggregated.hpp

Defined in header hpx/executors/parallel_executor_aggregated.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

hpx/executors/restricted_thread_pool_executor.hpp

Defined in header hpx/executors/restricted_thread_pool_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Typedefs

using **restricted_thread_pool_executor** = *restricted_policy_executor*<*hpx*::*launch*>

template<typename **Policy**>

class **restricted_policy_executor**

Public Types

using **execution_category** = typename *embedded_executor*::execution_category

Associate the parallel_execution_tag executor tag type as a default with this executor.

using **executor_parameters_type** = typename *embedded_executor*::executor_parameters_type

Public Functions

inline explicit **restricted_policy_executor**(*std*::size_t first_thread = 0, *std*::size_t num_threads = 1, *threads*::*thread_priority* priority = *threads*::*thread_priority*::*default_*, *threads*::*thread_stacksize* stacksize = *threads*::*thread_stacksize*::*default_*, *threads*::*thread_schedule_hint* schedulehint = {}, *std*::size_t hierarchical_threshold = *hierarchical_threshold_default_*)

Create a new parallel executor.

inline **restricted_policy_executor**(*restricted_policy_executor* const &other)

inline *restricted_policy_executor* &**operator=**(*restricted_policy_executor* const &rhs)

Private Types

using **embedded_executor** = *hpx*::*execution*::*parallel_policy_executor*<*Policy*>

Private Members

std::uint16_t **first_thread_**

mutable *std*::atomic<*std*::size_t> **os_thread_**

embedded_executor **exec_**

Private Static Attributes

static constexpr *std*::size_t **hierarchical_threshold_default_** = 6

hpx/executors/scheduler_executor.hpp

Defined in header hpx/executors/scheduler_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Functions

template<typename **BaseScheduler**>
explicit **scheduler_executor**(*BaseScheduler* &&sched) -> *scheduler_executor*<*std*::decay_t<*BaseScheduler*>>

template<typename **Tag**, typename **BaseScheduler**, typename **Property**>
auto **tag_invoke**(*Tag* tag, *scheduler_executor*<*BaseScheduler*> const &exec, *Property* &&prop) -> decltype(*scheduler_executor*<*BaseScheduler*>(*std*::declval<*Tag*>()(*std*::declval<*BaseScheduler*>(), *std*::declval<*Property*>())))

template<typename **Tag**, typename **BaseScheduler**>
auto **tag_invoke**(*Tag* tag, *scheduler_executor*<*BaseScheduler*> const &exec) -> decltype(*std*::declval<*Tag*>()(*std*::declval<*BaseScheduler*>()))

template<typename **BaseScheduler**>

struct **scheduler_executor**

namespace **parallel**

namespace **execution**

hpx/executors/sequenced_executor.hpp

Defined in header hpx/executors/sequenced_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

struct **sequenced_executor**

#include <sequenced_executor.hpp> A *sequential_executor* creates groups of sequential execution agents which execute in the calling thread. The sequential order is given by the lexicographical order of indices in the index space.

namespace **parallel**

namespace **execution**

hpx/executors/service_executors.hpp

Defined in header hpx/executors/service_executors.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

hpx/executors/std_execution_policy.hpp

Defined in header hpx/executors/std_execution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/executors/thread_pool_scheduler.hpp

Defined in header hpx/executors/thread_pool_scheduler.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **execution**

namespace **experimental**

Typedefs

using **thread_pool_scheduler** = *thread_pool_policy_scheduler*<*hpx*::*launch*>

Functions

template<typename **Tag**, typename **Policy**, typename **Property**>
auto **tag_invoke**(*Tag* tag, *thread_pool_policy_scheduler*<*Policy*> const &scheduler, *Property* &&prop) -> decltype(*std*::declval<*thread_pool_policy_scheduler*<*Policy*>>().policy(*std*::declval<*Tag*>()(*std*::decl
std::declval<*Property*>())), *thread_pool_policy_scheduler*<*Policy*>())

template<typename **Tag**, typename **Policy**>
auto **tag_invoke**(*Tag* tag, *thread_pool_policy_scheduler*<*Policy*> const &scheduler) -> decltype(*std*::declval<*Tag*>()(*std*::declval<*Policy*>()))

template<typename **Policy**>

struct **thread_pool_policy_scheduler**

Public Types

using **execution_category** = *std*::conditional_t<*std*::is_same_v<*Policy*, *launch*::sync_policy>, *sequenced_execution_tag*, *parallel_execution_tag*>

Public Functions

inline explicit constexpr **thread_pool_policy_scheduler**(*Policy* l = *experimental*::detail::get_default_scheduler_policy<*Policy*>::c

inline explicit **thread_pool_policy_scheduler**(*hpx*::*threads*::*thread_pool_base* *pool, *Policy* l = *experimental*::detail::get_default_scheduler_policy<*Policy*>::call()) noexcept

hpx/executors/datapar/execution_policy.hpp

Defined in header hpx/executors/datapar/execution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/executors/datapar/execution_policy_mappings.hpp

Defined in header hpx/executors/datapar/execution_policy_mappings.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

filesystem

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/modules/filesystem.hpp

Defined in header hpx/modules/filesystem.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

This file provides a compatibility layer using Boost.Filesystem for the C++17 filesystem library. It is *not* intended to be a complete compatibility layer. It only contains functions required by the HPX codebase. It also provides some functions only available in Boost.Filesystem when using C++17 filesystem.

namespace **hpx**

namespace **filesystem**

Functions

inline path **initial_path**()

inline *std*::string **basename**(path const &p)

inline path **canonical**(path const &p, path const &base)

inline path **canonical**(path const &p, path const &base, *std*::error_code &ec)

namespace **filesystem**

functional

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::bind, hpx::placeholders::_1, hpx::placeholders::_2, . . . , hpx::placeholders::_9

Defined in header hpx/functional.hpp[699].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

[699] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Functions

template<typename **F**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<!*traits*::is_action_v<*std*::decay_t<*F*>>>>
constexpr detail::bound<*std*::decay_t<*F*>, *util*::make_index_pack_t<sizeof...(*Ts*)>, *util*::decay_unwrap_t<*Ts*>...> **bind**(*F* &&f, *Ts*&&... vs)

The function template *bind* generates a forwarding call wrapper for *f*. Calling this wrapper is equivalent to invoking *f* with some of its arguments bound to *vs*.

Parameters

- **f** – Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments
- **vs** – list of arguments to bind, with the unbound arguments replaced by the placeholders _1, _2, _3… of namespace hpx::placeholders

Returns A function object of unspecified type *T*, for which

```
hpx::is_bind_expression<T>::value == true.
```

namespace **placeholders**

The *hpx::placeholders* namespace contains the placeholder objects [_1, …, _N] where N is an implementation defined maximum number.

When used as an argument in a *hpx::bind* expression, the placeholder objects are stored in the generated function object, and when that function object is invoked with unbound arguments, each placeholder _N is replaced by the corresponding Nth unbound argument.

The types of the placeholder objects are DefaultConstructible and CopyConstructible, their default copy/move constructors do not throw exceptions, and for any placeholder _N, the type hpx::is_placeholder<decltype(_N)> is defined, where hpx::is_placeholder<decltype(_N)> is derived from std::integral_constant<int, N>.

Variables

constexpr detail::placeholder<1> **_1** = {}

constexpr detail::placeholder<2> **_2** = {}

constexpr detail::placeholder<3> **_3** = {}

constexpr detail::placeholder<4> **_4** = {}

constexpr detail::placeholder<5> **_5** = {}

constexpr detail::placeholder<6> **_6** = {}

constexpr detail::placeholder<7> **_7** = {}

constexpr detail::placeholder<8> **_8** = {}

constexpr detail::placeholder<9> **_9** = {}

namespace **serialization**

Functions

template<typename **Archive**, typename **F**, typename ...**Ts**>
void **serialize**(*Archive* &ar, ::*hpx*::detail::bound<*F*, *Ts*...> &bound, unsigned int const version = 0)

template<typename **Archive**, *std*::size_t **I**>
constexpr void **serialize**(*Archive*&, ::*hpx*::detail::placeholder<*I*>&, unsigned int const = 0) noexcept

namespace **util**

Functions

template<typename F, typename... Ts> HPX_DEPRECATED_V (1, 8, "hpx::util::bind is deprecated, use hpx::bind instead") const expr decltype(auto) bind(F &&f

Variables

Ts && ts {returnhpx::bind(HPX_FORWARD(F, f), HPX_FORWARD(Ts, ts)...)

namespace **placeholders**

Functions

HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_1 is deprecated, use hpx::placeholders::_1 " "instead") inline const expr hpx

HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_2 is deprecated, use hpx::placeholders::_2 " "instead") inline const expr hpx

HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_3 is deprecated, use hpx::placeholders::_3 " "instead") inline const expr hpx

HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_4 is deprecated, use hpx::placeholders::_4 " "instead") inline const expr hpx

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_5 is deprecated,
use hpx::placeholders::_5 " "instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_6 is deprecated,
use hpx::placeholders::_6 " "instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_7 is deprecated,
use hpx::placeholders::_7 " "instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_8 is deprecated,
use hpx::placeholders::_8 " "instead") inline const expr hpx
```

```
HPX_DEPRECATED_V (1, 8, "hpx::placeholders::_9 is deprecated,
use hpx::placeholders::_9 " "instead") inline const expr hpx
```

hpx::bind_back

Defined in header hpx/functional.hpp[700].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

Functions

template<typename **F**, typename ...**Ts**>
constexpr *hpx*::detail::bound_back<*std*::decay_t<*F*>, *util*::make_index_pack_t<sizeof...(*Ts*)>, *util*::decay_unwrap_t<*Ts*>...> **bind**

Function templates `bind_back` generate a forwarding call wrapper for `f`. Calling this wrapper is equivalent to invoking `f` with its last `sizeof...`(Ts) parameters bound to `vs`.

Parameters

- **f** – Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments
- **vs** – list of the arguments to bind to the last `sizeof...`(Ts) parameters of `f`

Returns A function object of type `T` that is unspecified, except that the types of objects returned by two calls to *`hpx::bind_back`* with the same arguments are the same.

template<typename **F**>
constexpr *std*::decay_t<*F*> **bind_back**(*F* &&f)

namespace **serialization**

[700] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Functions

template<typename **Archive**, typename **F**, typename ...**Ts**>
void **serialize**(*Archive* &ar, ::*hpx*::detail::bound_back<*F*, *Ts*...> &bound, unsigned int const version = 0)

namespace **util**

Functions

```
template<typename F, typename... Ts>  HPX_DEPRECATED_V (1, 8,
"hpx::util::bind_back is deprecated,
use hpx::bind_back instead") const expr decltype(auto) bind_back(F &&f
```

hpx::bind_front

Defined in header hpx/functional.hpp[701].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

Functions

template<typename **F**, typename ...**Ts**>
constexpr detail::bound_front<*std*::decay_t<*F*>, *util*::make_index_pack_t<sizeof...(*Ts*)>, *util*::decay_unwrap_t<*Ts*>...> **bind_fro**

Function template `bind_front` generates a forwarding call wrapper for `f`. Calling this wrapper is equivalent to invoking `f` with its first `sizeof...`(Ts) parameters bound to `vs`.

Parameters

- **f** – Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments
- **vs** – list of the arguments to bind to the first or `sizeof...`(Ts) parameters of `f`

Returns A function object of type `T` that is unspecified, except that the types of objects returned by two calls to *`hpx::bind_front`* with the same arguments are the same.

template<typename **F**>
constexpr *std*::decay_t<*F*> **bind_front**(*F* &&f)

namespace **serialization**

[701] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Functions

template<typename **Archive**, typename **F**, typename ...**Ts**>
void **serialize**(*Archive* &ar, ::*hpx*::detail::bound_front<*F*, *Ts*...> &bound, unsigned int const version = 0)

namespace **util**

Functions

```
template<typename F, typename... Ts>  HPX_DEPRECATED_V (1, 8,
"hpx::util::bind_front is deprecated,
use hpx::bind_front instead") const expr decltype(auto) bind_front(F &&f
```

hpx::function

Defined in header hpx/functional.hpp[702].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_UTIL_REGISTER_FUNCTION_DECLARATION(Sig, F, Name)

HPX_UTIL_REGISTER_FUNCTION(Sig, F, Name)

namespace **hpx**

Top level namespace.

template<typename **Sig**, bool **Serializable** = false>

class **function**

#include <function.hpp> Class template *hpx::function* is a general-purpose polymorphic function wrapper. Instances of *hpx::function* can store, copy, and invoke any CopyConstructible Callable target — functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members. The stored callable object is called the target of *hpx::function*. If an *hpx::function* contains no target, it is called empty. Invoking the target of an empty *hpx::function* results in *hpx::error::bad_function_call* exception being thrown. *hpx::function* satisfies the requirements of CopyConstructible and CopyAssignable.

template<typename **R**, typename ...**Ts**, bool **Serializable**>

class **function**<*R*(*Ts*...), *Serializable*> : public *util*::detail::basic_function<*R*(*Ts*...), true, *Serializable*>

[702] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Public Types

using **result_type** = *R*

Public Functions

inline constexpr **function**(*std*::nullptr_t = nullptr) noexcept

function(*function* const&) = default

function(*function*&&) noexcept = default

function &**operator=**(*function* const&) = default

function &**operator=**(*function*&&) noexcept = default

~function() = default

template<typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable1** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *function*>>, typename **Enable2** = *std*::enable_if_t<*is_invocable_r_v*<*R*, *FD*&, *Ts*...>>>
inline **function**(*F* &&f)

template<typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable1** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *function*>>, typename **Enable2** = *std*::enable_if_t<*is_invocable_r_v*<*R*, *FD*&, *Ts*...>>>
inline *function* &**operator=**(*F* &&f)

Private Types

using **base_type** = *util*::detail::basic_function<*R*(*Ts*...), true, *Serializable*>

namespace **distributed**

Typedefs

template<typename **Sig**>

using **function** = *hpx*::*function*<*Sig*, true>

namespace **util**

Typedefs

typedef *hpx*::*function*<Sig, Serializable> **instead**

hpx::function_ref

Defined in header hpx/functional.hpp[703].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

template<typename **Sig**>

class **function_ref**

#include <function_ref.hpp> `function_ref` class is a vocabulary type with reference semantics for passing entities to call.

An example use case that benefits from higher-order functions is `retry(n,f)` which attempts to call `f` up to `n` times synchronously until success. This example might model the real-world scenario of repeatedly querying a flaky web service.

```
using payload = std::optional< /* ... */ >;
// Repeatedly invokes `action` up to `times` repetitions.
// Immediately returns if `action` returns a valid `payload`.
// Returns `std::nullopt` otherwise.
payload retry(size_t times, /* ????? */ action);
```

The passed-in action should be a callable entity that takes no arguments and returns a payload. This can be done with function pointers, `hpx::function` or a template but it is much simpler with `function_ref` as seen below:

```
payload retry(size_t times, function_ref<payload()> action);
```

template<typename **R**, typename ...**Ts**>

class **function_ref**<*R*(*Ts*...)>

Public Functions

template<typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *function_ref*> && *is_invocable_r_v*<*R*, *F*&, *Ts*...>>>
inline **function_ref**(*F* &&f)

inline **function_ref**(*function_ref* const &other) noexcept

template<typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *function_ref*> && *is_invocable_r_v*<*R*, *F*&, *Ts*...>>>
inline *function_ref* &**operator=**(*F* &&f)

[703] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

inline *function_ref* &**operator=**(*function_ref* const &other) noexcept

template<typename **F**, typename **T** = *std*::remove_reference_t<*F*>, typename **Enable** = *std*::enable_if_t<!*std*::is_pointer_v<*T*>>>
inline void **assign**(*F* &&f)

template<typename **T**>
inline void **assign**(*std*::reference_wrapper<*T*> f_ref) noexcept

template<typename **T**>
inline void **assign**(*T* *f_ptr) noexcept

inline void **swap**(*function_ref* &f) noexcept

inline *R* **operator()**(*Ts*... vs) const

inline *std*::size_t **get_function_address**() const

inline char const ***get_function_annotation**() const

inline *util*::itt::string_handle **get_function_annotation_itt**() const

Protected Attributes

R (***vptr**)(void*, *Ts*&&...)

void ***object**

Private Types

using **VTable** = *util*::detail::function_ref_vtable<*R*(*Ts*...)>

Private Static Functions

template<typename **T**>
static inline constexpr *VTable* const ***get_vtable**() noexcept

namespace **util**

hpx::invoke

Defined in header hpx/functional.hpp[704].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

[704] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Defines

`HPX_INVOKE_R`(R, F, ...)

namespace **`hpx`**

Top level namespace.

Functions

template<typename **F**, typename ...**Ts**>
constexpr *util*::invoke_result_t<*F*, *Ts*&&...> **`invoke`**(*F* &&f, *Ts*&&... vs) noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(*F*, *f*), HPX_FORWARD(*Ts*, *vs*)...)))

Invokes the given callable object f with the content of the argument pack vs

> **Note:** This function is similar to `std::invoke` (C++17)

Parameters

- **`f`** – Requires to be a callable object. If f is a member function pointer, the first argument in the pack will be treated as the callee (this object).
- **`vs`** – An arbitrary pack of arguments

Throws `std::exception` – like objects thrown by call to object f with the argument types vs.

Returns The result of the callable object when it's called with the given argument types.

template<typename **R**, typename **F**, typename ...**Ts**>
constexpr *R* **`invoke_r`**(*F* &&f, *Ts*&&... vs) noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(*F*, *f*), HPX_FORWARD(*Ts*, *vs*)...)))

Invokes the given callable object f with the content of the argument pack vs

> **Note:** This function is similar to `std::invoke` (C++17)

Parameters

- **`f`** – Requires to be a callable object. If f is a member function pointer, the first argument in the pack will be treated as the callee (this object).
- **`vs`** – An arbitrary pack of arguments

Throws `std::exception` – like objects thrown by call to object f with the argument types vs.

Template Parameters **`R`** – The result type of the function when it's called with the content of the given argument types vs.

Returns The result of the callable object when it's called with the given argument types.

namespace **`functional`**

struct **`invoke`**

Public Functions

template<typename **F**, typename ...**Ts**>
inline constexpr *util*::invoke_result_t<*F*, *Ts*&&...> **operator()**(*F* &&f, *Ts*&&... vs) const noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(*F*, *f*), HPX_FORWARD(*Ts*, *vs*)...)))

template<typename **R**>

struct **invoke_r**

Public Functions

template<typename **F**, typename ...**Ts**>
inline constexpr *R* **operator()**(*F* &&f, *Ts*&&... vs) const noexcept(noexcept(HPX_INVOKE(HPX_FORWARD(*F*, *f*), HPX_FORWARD(*Ts*, *vs*)...)))

hpx::invoke_fused, hpx::invoke_fused_r

Defined in header hpx/functional.hpp[705].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

Functions

template<typename **F**, typename **Tuple**>
constexpr detail::invoke_fused_result_t<*F*, *Tuple*> **invoke_fused**(*F* &&f, *Tuple* &&t) noexcept(noexcept(detail::invoke_fused_impl(detail::fused_index_
HPX_FORWARD(*F*, *f*),
HPX_FORWARD(*Tuple*, *t*))))

Invokes the given callable object f with the content of the sequenced type t (tuples, pairs).

Note: This function is similar to `std::apply` (C++17). The difference between `hpx::invoke` and `hpx::invoke_fused` is that the later unpacks the tuples while the former cannot. Turning a tuple into a parameter pack is not a trivial operation which makes `hpx::invoke_fused` rather useful.

Parameters

- **f** – Must be a callable object. If f is a member function pointer, the first argument in the sequenced type will be treated as the callee (this object).
- **t** – A type whose contents are accessible through a call to hpx::get.

Throws `std::exception` – like objects thrown by call to object f with the arguments contained in the sequenceable type t.

[705] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Returns The result of the callable object when it's called with the content of the given sequenced type.

template<typename **R**, typename **F**, typename **Tuple**>
constexpr *R* **invoke_fused_r**(*F* &&f, *Tuple* &&t) noexcept(noexcept(detail::invoke_fused_impl(detail::fused_index_pack_t<*Tuple*>{}, HPX_FORWARD(*F*, *f*), HPX_FORWARD(*Tuple*, *t*))))

Invokes the given callable object f with the content of the sequenced type t (tuples, pairs).

Note: This function is similar to `std::apply` (C++17). The difference between `hpx::invoke` and `hpx::invoke_fused` is that the later unpacks the tuples while the former cannot. Turning a tuple into a parameter pack is not a trivial operation which makes `hpx::invoke_fused` rather useful.

Note: The difference between `hpx::invoke_fused` and `hpx::invoke_fused_r` is that the later allows to specify the return type as well.

Parameters

- **f** – Must be a callable object. If f is a member function pointer, the first argument in the sequenced type will be treated as the callee (this object).
- **t** – A type whose contents are accessible through a call to hpx::get.

Throws `std::exception` – like objects thrown by call to object f with the arguments contained in the sequenceable type t.

Template Parameters **R** – The result type of the function when it's called with the content of the given sequenced type.

Returns The result of the callable object when it's called with the content of the given sequenced type.

hpx::mem_fn

Defined in header hpx/functional.hpp[706].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

[706] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Functions

template<typename **M**, typename **C**>
constexpr detail::mem_fn<*M* *C*::*> **mem_fn**(*M* *C*::* pm) noexcept

Function template *hpx::mem_fn* generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a *hpx::mem_fn*.

Parameters **pm** – pointer to member that will be wrapped

Returns a call wrapper of unspecified type with the following member:

```
template <typename... Ts>
constexpr typename util::invoke_result<MemberPointer, Ts...>::type
operator()(Ts&&... vs) noexcept;
```

Let `fn` be the call wrapper returned by a call to *hpx::mem_fn* with a pointer to member pm. Then the expression `fn(t,a2,...,aN)` is equivalent to `HPX_INVOKE(pm,t,a2,...,aN)`. Thus, the return type of operator() is `std::result_of<decltype(pm)(Ts&&...)>::type` or equivalently `std::invoke_result_t<decltype(pm),Ts&&...>`, and the value in `noexcept` specifier is equal to `std::is_nothrow_invocable_v<decltype(pm),Ts&&...>)` . Each argument in `vs` is perfectly forwarded, as if by `std::forward<Ts>(vs)...` .

template<typename **R**, typename **C**, typename ...**Ps**>
constexpr detail::mem_fn<*R* (*C*::*)(*Ps*...)> **mem_fn**(*R* (*C*::* pm)(*Ps*...)) noexcept

Function template *hpx::mem_fn* generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a *hpx::mem_fn*.

Parameters **pm** – pointer to member that will be wrapped

Returns a call wrapper of unspecified type with the following member:

```
template <typename... Ts>
constexpr typename util::invoke_result<MemberPointer, Ts...>::type
operator()(Ts&&... vs) noexcept;
```

Let `fn` be the call wrapper returned by a call to *hpx::mem_fn* with a pointer to member pm. Then the expression `fn(t,a2,...,aN)` is equivalent to `HPX_INVOKE(pm,t,a2,...,aN)`. Thus, the return type of operator() is `std::result_of<decltype(pm)(Ts&&...)>::type` or equivalently `std::invoke_result_t<decltype(pm),Ts&&...>`, and the value in `noexcept` specifier is equal to `std::is_nothrow_invocable_v<decltype(pm),Ts&&...>)` . Each argument in `vs` is perfectly forwarded, as if by `std::forward<Ts>(vs)...` .

template<typename **R**, typename **C**, typename ...**Ps**>
constexpr detail::mem_fn<*R* (*C*::*)(*Ps*...) const> **mem_fn**(*R* (*C*::* pm)(*Ps*...) const) noexcept

Function template *hpx::mem_fn* generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a *hpx::mem_fn*.

Parameters **pm** – pointer to member that will be wrapped

Returns a call wrapper of unspecified type with the following member:

```
template <typename... Ts>
constexpr typename util::invoke_result<MemberPointer, Ts...>::type
operator()(Ts&&... vs) noexcept;
```

Let `fn` be the call wrapper returned by a call to *hpx::mem_fn* with a pointer to member `pm`. Then the expression `fn(t,a2,...,aN)` is equivalent to `HPX_INVOKE(pm,t,a2,...,aN)`. Thus, the return type of operator() is `std::result_of<decltype(pm)(Ts&&...)>::type` or equivalently `std::invoke_result_t<decltype(pm),Ts&&...>`, and the value in `noexcept` specifier is equal to `std::is_nothrow_invocable_v<decltype(pm),Ts&&...>)` . Each argument in `vs` is perfectly forwarded, as if by `std::forward<Ts>(vs)...` .

hpx::move_only_function

Defined in header hpx/functional.hpp[707].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_UTIL_REGISTER_UNIQUE_FUNCTION_DECLARATION(Sig, F, Name)

HPX_UTIL_REGISTER_UNIQUE_FUNCTION(Sig, F, Name)

namespace **hpx**

Top level namespace.

template<typename **Sig**, bool **Serializable** = false>

class **move_only_function**

#include <move_only_function.hpp> Class template *hpx::move_only_function* is a general-purpose polymorphic function wrapper. *hpx::move_only_function* objects can store and invoke any constructible (not required to be move constructible) Callable target — functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to member objects.

The stored callable object is called the target of *hpx::move_only_function*. If an *hpx::move_only_function* contains no target, it is called empty. Unlike *hpx::function*, invoking an empty *hpx::move_only_function* results in undefined behavior.

hpx::move_only_functions supports every possible combination of cv-qualifiers, ref-qualifiers, and noexcept-specifiers not including volatile provided in its template parameter. These qualifiers and specifier (if any) are added to its operator(). *hpx::move_only_function* satisfies the requirements of MoveConstructible and MoveAssignable, but does not satisfy CopyConstructible or CopyAssignable.

template<typename **R**, typename ...**Ts**, bool **Serializable**>

class **move_only_function**<*R*(*Ts*...), *Serializable*> : public *util*::detail::basic_function<*R*(*Ts*...), false, *Serializable*>

[707] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Public Types

using **result_type** = *R*

Public Functions

inline constexpr **move_only_function**(*std*::nullptr_t = nullptr) noexcept

move_only_function(*move_only_function* const&) = delete

move_only_function(*move_only_function*&&) noexcept = default

move_only_function &**operator=**(*move_only_function* const&) = delete

move_only_function &**operator=**(*move_only_function*&&) noexcept = default

~move_only_function() = default

template<typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable1** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *move_only_function*>>, typename **Enable2** = *std*::enable_if_t<*is_invocable_r_v*<*R*, *FD*&, *Ts*...>>>
inline **move_only_function**(*F* &&f)

template<typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable1** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *move_only_function*>>, typename **Enable2** = *std*::enable_if_t<*is_invocable_r_v*<*R*, *FD*&, *Ts*...>>>
inline *move_only_function* &**operator=**(*F* &&f)

Private Types

using **base_type** = *util*::detail::basic_function<*R*(*Ts*...), false, *Serializable*>

namespace **distributed**

Typedefs

template<typename **Sig**>

using **move_only_function** = *hpx*::*move_only_function*<*Sig*, true>

namespace **util**

hpx::experimental::scope_exit

Defined in header hpx/experimental/scope.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

namespace **experimental**

Functions

template<typename **F**>
auto **scope_exit**(*F* &&f)

The class template scope_exit is a general-purpose scope guard intended to call its exit function when a scope is exited.

Template Parameters **F** – type of stored exit function

Parameters **f** – stored exit function

hpx::experimental::scope_fail

Defined in header hpx/experimental/scope.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

namespace **experimental**

Functions

template<typename **F**>
auto **scope_fail**(*F* &&f)

The class template scope_fail is a general-purpose scope guard intended to call its exit function when a scope is exited via an exception.

Template Parameters **F** – type of stored exit function

Parameters **f** – stored exit function

hpx::experimental::scope_success

Defined in header hpx/experimental/scope.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

namespace **experimental**

Functions

template<typename **F**>
auto **scope_success**(*F* &&f)

The class template scope_success is a general-purpose scope guard intended to call its exit function when a scope is normally exited.

Template Parameters **F** – type of stored exit function

Parameters **f** – stored exit function

hpx::is_bind_expression

Defined in header hpx/functional.hpp[708].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

Variables

template<typename **T**>

constexpr bool **is_bind_expression_v** = *is_bind_expression*<*T*>::value

template<typename **T**>

struct **is_bind_expression** : public *std*::is_bind_expression<*T*>

#include <is_bind_expression.hpp> If `T` is the type produced by a call to `hpx::bind`, this template is derived from `std::true_type`. For any other type, this template is derived from `std::false_type`.

This template may be specialized for a user-defined type `T` to implement *UnaryTypeTrait* with base characteristic of `std::true_type` to indicate that `T` should be treated by `hpx::bind` as if it were the type of a bind subexpression: when a bind-generated function object is invoked, a bound argument of this type will be invoked as a function object and will be given all the unbound arguments passed to the bind-generated object.

Subclassed by *hpx::is_bind_expression< T const >*

template<typename **T**>

[708] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

struct **is_bind_expression**<*T* const> : public *hpx*::*is_bind_expression*<*T*>

namespace **traits**

Typedefs

typedef *hpx*::*is_bind_expression*<T> **instead**

Functions

template<typename T> HPX_DEPRECATED_V (1, 8, "hpx::traits::is_bind_expression_v is deprecated, use " "hpx::is_bind_expression_v instead") inline const expr bool is_bind_expression_v

hpx::is_placeholder

Defined in header hpx/functional.hpp[709].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level namespace.

template<typename **T**>

struct **is_placeholder**

#include <is_placeholder.hpp> If T is a standard, Boost, or HPX placeholder (_1, _2, _3, …) then this template is derived from std::integral_constant<int,1>, std::integral_constant<int, 2>, std::integral_constant<int,3>, respectively. Otherwise, it is derived from std::integral_constant<int,0>.

The template may be specialized for any user-defined T type: the specialization must satisfy *UnaryTypeTrait* with base characteristic of std::integral_constant<int,N> with N>0 to indicate that T should be treated as N'th placeholder type. *hpx::bind* uses *hpx::is_placeholder* to detect placeholders for unbound arguments.

namespace **traits**

[709] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Functions

`template<typename T> HPX_DEPRECATED_V (1, 8, "hpx::traits::is_placeholder_v is deprecated, use " "hpx::is_placeholder_v instead") inline const expr bool is_placeholder_v`

futures

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::future, hpx::shared_future, hpx::make_future, hpx::make_shared_future, hpx::make_ready_future, hpx::make_ready_future_alloc, hpx::make_ready_future_at, hpx::make_ready_future_after, hpx::make_exceptional_future

Defined in header hpx/future.hpp[710].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_MAKE_EXCEPTIONAL_FUTURE(T, errorcode, f, msg)

namespace **hpx**

Top level HPX namespace.

Functions

template<typename **R**, typename **U**>
hpx::*future*<*R*> **make_future**(*hpx*::*future*<*U*> &&f)

Converts any future of type U to any other future of type R based on an existing conversion path from U to R.

template<typename **R**, typename **U**, typename **Conv**>
hpx::*future*<*R*> **make_future**(*hpx*::*future*<*U*> &&f, *Conv* &&conv)

Converts any future of type U to any other future of type R based on a given conversion function: R conv(U).

template<typename **R**, typename **U**>
hpx::*future*<*R*> **make_future**(*hpx*::*shared_future*<*U*> f)

Converts any *shared_future* of type U to any other future of type R based on an existing conversion path from U to R.

template<typename **R**, typename **U**, typename **Conv**>
hpx::*future*<*R*> **make_future**(*hpx*::*shared_future*<*U*> f, *Conv* &&conv)

Converts any future of type U to any other future of type R based on an existing conversion path from U to R.

template<typename **R**>

[710] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

hpx::shared_future<R> **make_shared_future**(hpx::future<R> &&f) noexcept

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

template<typename **R**>
hpx::shared_future<R> &**make_shared_future**(hpx::shared_future<R> &f) noexcept

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

template<typename **R**>
hpx::shared_future<R> &&**make_shared_future**(hpx::shared_future<R> &&f) noexcept

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

template<typename **R**>
hpx::shared_future<R> const &**make_shared_future**(hpx::shared_future<R> const &f) noexcept

Converts any future or *shared_future* of type T to a corresponding *shared_future* of type T.

template<typename **T**, typename **Allocator**, typename ...**Ts**>
std::enable_if_t<std::is_constructible_v<T, Ts&&...> || std::is_void_v<T>, future<T>> **make_ready_future_alloc**(Allocator const &a, Ts&&... ts)

Creates a pre-initialized future object with allocator (extension)

template<typename **T**, typename ...**Ts**>
std::enable_if_t<std::is_constructible_v<T, Ts&&...> || std::is_void_v<T>, future<T>> **make_ready_future**(Ts&&... ts)

The function creates a shared state that is immediately ready and returns a future associated with that shared state. For the returned future, valid() == true and is_ready() == true.

template<int **DeductionGuard** = 0, typename **Allocator**, typename **T**>
future<hpx::util::decay_unwrap_t<T>> **make_ready_future_alloc**(Allocator const &a, T &&init)

template<int **DeductionGuard** = 0, typename **T**>
future<hpx::util::decay_unwrap_t<T>> **make_ready_future**(T &&init)

The function creates a shared state that is immediately ready and returns a future associated with that shared state. For the returned future, valid() == true and is_ready() == true.

template<typename **T**>
future<T> **make_exceptional_future**(std::exception_ptr const &e)

Creates a pre-initialized future object which holds the given error (extension)

template<typename **T**, typename **E**>
future<T> **make_exceptional_future**(E e)

Creates a pre-initialized future object which holds the given error (extension)

template<int **DeductionGuard** = 0, typename **T**>
future<hpx::util::decay_unwrap_t<T>> **make_ready_future_at**(hpx::chrono::steady_time_point const &abs_time, T &&init)

Creates a pre-initialized future object which gets ready at a given point in time (extension)

template<int **DeductionGuard** = 0, typename **T**>
future<hpx::util::decay_unwrap_t<T>> **make_ready_future_after**(hpx::chrono::steady_duration const &rel_time, T &&init)

Creates a pre-initialized future object which gets ready after a given point in time (extension)

template<typename **Allocator**>

inline *future*<void> **make_ready_future_alloc**(*Allocator* const &a)

future<void> **make_ready_future**()

The function creates a shared state that is immediately ready and returns a future associated with that shared state. For the returned future, valid() == true and is_ready() == true.

inline *future*<void> **make_ready_future_at**(*hpx*::*chrono*::steady_time_point const &abs_time)

Creates a pre-initialized future object which gets ready at a given point in time (extension)

inline *future*<void> **make_ready_future_after**(*hpx*::*chrono*::steady_duration const &rel_time)

Creates a pre-initialized future object which gets ready after a given point in time (extension)

template<typename **R**>

class **future** : public *hpx*::*lcos*::detail::future_base<*future*<*R*>, *R*>

#include <future_fwd.hpp> The class template *hpx::future* provides a mechanism to access the result of asynchronous operations:

- An asynchronous operation (created via hpx::async, *hpx::packaged_task*, or *hpx::promise*) can provide a *hpx::future* object to the creator of that asynchronous operation.
- The creator of the asynchronous operation can then use a variety of methods to query, wait for, or extract a value from the *hpx::future*. These methods may block if the asynchronous operation has not yet provided a value.
- When the asynchronous operation is ready to send a result to the creator, it can do so by modifying shared state (e.g. hpx::promise::set_value) that is linked to the creator's *hpx::future*. Note that *hpx::future* references shared state that is not shared with any other asynchronous return objects (as opposed to *hpx::shared_future*).

Public Types

using **result_type** = *R*

using **shared_state_type** = typename *base_type*::shared_state_type

Public Functions

constexpr **future**() noexcept = default

future(*future* &&other) noexcept = default

future(*future* const &other) noexcept = delete

inline **future**(*future*<*future*> &&other) noexcept

inline **future**(*future*<*shared_future*<*R*>> &&other) noexcept

template<typename **T**>
inline **future**(*future*<*T*> &&other, *std*::enable_if_t<*std*::is_void_v<*R*> && !*traits*::is_future_v<*T*>, *T*>* = nullptr) noexcept

~future() = default

future &**operator=**(*future* &&other) noexcept = default

future &**operator=**(*future* const &other) noexcept = delete

inline *shared_future*<*R*> **share**() noexcept

inline *hpx*::*traits*::future_traits<*future*>::result_type **get**()

inline *hpx*::*traits*::future_traits<*future*>::result_type **get**(*error_code* &ec)

template<typename **F**>
inline decltype(auto) **then**(*F* &&f, *error_code* &ec = *throws*)

Attaches a continuation to **this*. The behavior is undefined if **this* has no associated shared state (i.e., *valid()==false*).

In cases where *decltype(func(*this))* is *future<R>*, the resulting type is *future<R>* instead of *future<future<R>>*. Effects:

- The continuation is called when the object's shared state is ready (has a value or exception stored).
- The continuation launches according to the specified launch policy or executor.
- When the executor or launch policy is not provided the continuation inherits the parent's launch policy or executor.
- If the parent was created with *std::promise* or with a *packaged_task* (has no associated launch policy), the continuation behaves the same as the third overload with a policy argument of *launch::async* | *launch::deferred* and the same argument for *func*.
- If the parent has a policy of *launch::deferred* and the continuation does not have a specified launch policy or scheduler, then the parent is filled by immediately calling .wait(), and the policy of the antecedent is *launch::deferred*

Note: Postcondition:

- The future object is moved to the parameter of the continuation function.
- valid() == false on original future object immediately after it returns.

Template Parameters

- **F** – The type of the function/function object to use (deduced). F must meet requirements of *MoveConstructible*.
- **error_code** – The type of error code.

Parameters

- **f** – A continuation to be attached.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns An object of type *future<decltype(func(*this))>* that refers to the shared state created by the continuation.

template<typename **T0**, typename **F**>
inline decltype(auto) **then**(*T0* &&t0, *F* &&f, *error_code* &ec = *throws*)

Attaches a continuation to **this*. The behavior is undefined if **this* has no associated shared state (i.e., *valid()==false*). \copydetail *hpx::future::then*(F&& f, error_code& ec = throws)

Note: Postcondition:

- The future object is moved to the parameter of the continuation function.
- valid() == false on original future object immediately after it returns.

Template Parameters

- **T0** – The type of executor or launch policy.

- **F** – The type of the function/function object to use (deduced). F must meet requirements of *MoveConstructible*.
- **error_code** – The type of error code.

Parameters

- **t0** – The executor or launch policy to be used.
- **f** – A continuation to be attached.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns An object of type *future<decltype(func(*this))>* that refers to the shared state created by the continuation.

template<typename **Allocator**, typename **F**>
inline auto **then_alloc**(*Allocator* const &alloc, *F* &&f, *error_code* &ec = *throws*) -> decltype(*base_type*::then_alloc(*alloc*, HPX_MOVE(*this), HPX_FORWARD(*F*, *f*), *ec*))

Private Types

using **base_type** = *lcos*::detail::future_base<*future*<*R*>, *R*>

Private Functions

inline explicit **future**(*hpx*::intrusive_ptr<*shared_state_type*> const &state)

inline explicit **future**(*hpx*::intrusive_ptr<*shared_state_type*> &&state)

template<typename **SharedState**>
inline explicit **future**(*hpx*::intrusive_ptr<*SharedState*> const &state)

Friends

friend struct hpx::traits::future_access

template<typename **R**>

class **shared_future** : public *hpx*::*lcos*::detail::future_base<*shared_future*<*R*>, *R*>

#include <future_fwd.hpp> The class template *hpx::shared_future* provides a mechanism to access the result of asynchronous operations, similar to *hpx::future*, except that multiple threads are allowed to wait for the same shared state. Unlike *hpx::future*, which is only moveable (so only one instance can refer to any particular asynchronous result), *hpx::shared_future* is copyable and multiple shared future objects may refer to the same shared state. Access to the same shared state from multiple threads is safe if each thread does it through its own copy of a *shared_future* object.

Public Types

using **result_type** = *R*

using **shared_state_type** = typename *base_type*::shared_state_type

Public Functions

constexpr **shared_future**() noexcept = default

shared_future(*shared_future* const &other) = default

shared_future(*shared_future* &&other) noexcept = default

inline **shared_future**(*future*<*R*> &&other) noexcept

inline **shared_future**(*future*<*shared_future*> &&other) noexcept

template<typename **T**>
inline **shared_future**(*shared_future*<*T*> const &other, *std*::enable_if_t<*std*::is_void_v<*R*> && !*traits*::is_future_v<*T*>, *T*>* = nullptr)

~shared_future() = default

shared_future &**operator=**(*shared_future* const &other) = default

shared_future &**operator=**(*shared_future* &&other) noexcept = default

inline *hpx*::*traits*::future_traits<*shared_future*>::result_type **get**() const

inline *hpx*::*traits*::future_traits<*shared_future*>::result_type **get**(*error_code* &ec) const

template<typename **F**>
inline decltype(auto) **then**(*F* &&f, *error_code* &ec = *throws*) const

Attaches a continuation to **this*. The behavior is undefined if **this* has no associated shared state (i.e., *valid()==false*).

In cases where *decltype(func(*this))* is *future<R>*, the resulting type is *future<R>* instead of *future<future<R>>*. Effects:

- The continuation is called when the object's shared state is ready (has a value or exception stored).
- The continuation launches according to the specified launch policy or executor.
- When the executor or launch policy is not provided the continuation inherits the parent's launch policy or executor.
- If the parent was created with *std::promise* or with a *packaged_task* (has no associated launch policy), the continuation behaves the same as the third overload with a policy argument of *launch::async* | *launch::deferred* and the same argument for *func*.
- If the parent has a policy of *launch::deferred* and the continuation does not have a specified launch policy or scheduler, then the parent is filled by immediately calling .wait(), and the policy of the antecedent is *launch::deferred*

Note: Postcondition:

- The future object is moved to the parameter of the continuation function.
- valid() == false on original future object immediately after it returns.

Template Parameters

- **F** – The type of the function/function object to use (deduced). F must meet requirements of *MoveConstructible*.
- **error_code** – The type of error code.

Parameters

- **f** – A continuation to be attached.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns An object of type *future<decltype(func(*this))>* that refers to the shared state created by the continuation.

template<typename **T0**, typename **F**>
inline decltype(auto) **then**(*T0* &&t0, *F* &&f, *error_code* &ec = *throws*) const

Attaches a continuation to **this*. The behavior is undefined if **this* has no associated shared state (i.e., *valid()==false*). \copydetail *hpx::future::then*(F&& f, error_code& ec = throws)

Note: Postcondition:

- The future object is moved to the parameter of the continuation function.
- valid() == false on original future object immediately after it returns.

Template Parameters

- **T0** – The type of executor or launch policy.
- **F** – The type of the function/function object to use (deduced). F must meet requirements of *MoveConstructible*.
- **error_code** – The type of error code.

Parameters

- **t0** – The executor or launch policy to be used.
- **f** – A continuation to be attached.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns An object of type *future<decltype(func(*this))>* that refers to the shared state created by the continuation.

template<typename **Allocator**, typename **F**>
inline auto **then_alloc**(*Allocator* const &alloc, *F* &&f, *error_code* &ec = *throws*) -> decltype(*base_type*::then_alloc(*alloc*, HPX_MOVE(*this), HPX_FORWARD(*F*, *f*), *ec*))

Private Types

using **base_type** = *lcos*::detail::future_base<*shared_future*<*R*>, *R*>

Private Functions

inline explicit **shared_future**(*hpx*::intrusive_ptr<*shared_state_type*> const &state)

inline explicit **shared_future**(*hpx*::intrusive_ptr<*shared_state_type*> &&state)

template<typename **SharedState**>
inline explicit **shared_future**(*hpx*::intrusive_ptr<*SharedState*> const &state)

Friends

friend struct hpx::traits::future_access

namespace **lcos**

Functions

template<typename R, typename U> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_future is deprecated. Use hpx::make_future instead.") hpx

template<typename R, typename U, typename Conv> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_future is deprecated. Use hpx::make_future instead.") hpx

template<typename T, typename Allocator, typename... Ts> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future_alloc is deprecated. Use " "hpx::make_ready_future_alloc instead.") std

template<typename T, typename... Ts> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future is deprecated. Use " "hpx::make_ready_future instead.") std

template<typename T> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_exceptional_future is deprecated. Use " "hpx::make_exceptional_future instead.") hpx

template<typename T, typename E> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_exceptional_future is deprecated. Use " "hpx::make_exceptional_future instead.") hpx

template<int DeductionGuard = 0, typename T> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future_at is deprecated. Use " "hpx::make_ready_future_at instead.") hpx

template<int DeductionGuard = 0, typename T> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future_after is deprecated. Use " "hpx::make_ready_future_after instead.") hpx

template<typename Allocator> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future_alloc is deprecated. Use " "hpx::make_ready_future_alloc instead.") hpx

template<typename T> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future is deprecated. Use " "hpx::make_ready_future instead.") std

template<typename T> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future_at is deprecated. Use " "hpx::make_ready_future_at instead.") std

template<typename T> HPX_DEPRECATED_V (1, 8, "hpx::lcos::make_ready_future_after is deprecated. Use " "hpx::make_ready_future_after instead.") std

namespace **serialization**

Functions

template<typename **Archive**, typename **T**>
void **serialize**(*Archive* &ar, ::*hpx*::*future*<*T*> &f, unsigned version)

template<typename **Archive**, typename **T**>
void **serialize**(*Archive* &ar, ::*hpx*::*shared_future*<*T*> &f, unsigned version)

hpx/futures/future_fwd.hpp

Defined in header hpx/futures/future_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

template<typename **R**>

class **future** : public *hpx*::*lcos*::detail::future_base<*future*<*R*>, *R*>

#include <future_fwd.hpp>

template<typename **R**>

class **shared_future** : public *hpx*::*lcos*::detail::future_base<*shared_future*<*R*>, *R*>

#include <future_fwd.hpp>

namespace **lcos**

Typedefs

typedef *hpx*::*future*<R> **instead**

namespace **lcos**

hpx::packaged_task

Defined in header hpx/future.hpp[711].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **Sig**, typename **Allocator**>

struct **uses_allocator**<*hpx*::*packaged_task*<*Sig*>, *Allocator*> : public true_type

namespace **hpx**

Top level HPX namespace.

template<typename **Sig**>

class **packaged_task**

#include <packaged_task.hpp> The class template *hpx::packaged_task* wraps any Callable` target (function, lambda expression, bind expression, or another function object) so that it can be invoked asynchronously. Its return value or exception thrown is stored in a shared state which can be accessed through *hpx::future* objects. Just like hpx::function, *hpx::packaged_task* is a polymorphic, allocator-aware container: the stored callable target may be allocated on heap or with a provided allocator.

template<typename **R**, typename ...**Ts**>

class **packaged_task**<*R*(*Ts*...)>

Public Functions

packaged_task() = default

template<typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *packaged_task*> && *is_invocable_r_v*<*R*, *FD*&, *Ts*...>>>
inline explicit **packaged_task**(*F* &&f)

template<typename **Allocator**, typename **F**, typename **FD** = *std*::decay_t<*F*>, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*FD*, *packaged_task*> && *is_invocable_r_v*<*R*, *FD*&, *Ts*...>>>
inline explicit **packaged_task**(*std*::allocator_arg_t, *Allocator* const &a, *F* &&f)

packaged_task(*packaged_task* const &rhs) noexcept = delete

packaged_task(*packaged_task* &&rhs) noexcept = default

packaged_task &**operator=**(*packaged_task* const &rhs) noexcept = delete

packaged_task &**operator=**(*packaged_task* &&rhs) noexcept = default

[711] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

inline void **swap**(*packaged_task* &rhs) noexcept

inline void **operator()**(*Ts*... ts)

inline *hpx*::*future*<*R*> **get_future**(*error_code* &ec = *throws*)

inline bool **valid**() const noexcept

inline void **reset**(*error_code* &ec = *throws*)

inline void **set_exception**(*std*::exception_ptr const &e)

Private Types

using **function_type** = *hpx*::*move_only_function*<*R*(*Ts*...)>

Private Members

function_type **function_**

hpx::*promise*<*R*> **promise_**

namespace **lcos**

namespace **local**

Typedefs

typedef *hpx*::*packaged_task*<Sig> **instead**

namespace **std**

Functions

template<typename **Sig**>
void **swap**(*hpx*::*packaged_task*<*Sig*> &lhs, *hpx*::*packaged_task*<*Sig*> &rhs) noexcept

template<typename Sig, typename Allocator> packaged_task< Sig >, Allocator > : public true_type

hpx::promise

Defined in header hpx/future.hpp[712].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **R**, typename **Allocator**>

struct **uses_allocator**<*hpx*::*promise*<*R*>, *Allocator*> : public true_type

namespace **hpx**

Top level HPX namespace.

template<typename **R**>

class **promise** : public *hpx*::detail::promise_base<*R*>

#include <promise.hpp> The class template *hpx::promise* provides a facility to store a value or an exception that is later acquired asynchronously via a *hpx::future* object created by the *hpx::promise* object. Note that the *hpx::promise* object is meant to be used only once. Each promise is associated with a shared state, which contains some state information and a result which may be not yet evaluated, evaluated to a value (possibly void) or evaluated to an exception. A promise may do three things with the shared state:

- make ready: the promise stores the result or the exception in the shared state. Marks the state ready and unblocks any thread waiting on a future associated with the shared state.
- release: the promise gives up its reference to the shared state. If this was the last such reference, the shared state is destroyed. Unless this was a shared state created by hpx::async which is not yet ready, this operation does not block.
- abandon: the promise stores the exception of type hpx::future_error with error code hpx::error::broken_promise, makes the shared state ready, and then releases it. The promise is the "push" end of the promise-future communication channel: the operation that stores a value in the shared state synchronizes-with (as defined in hpx::memory_order) the successful return from any function that is waiting on the shared state (such as hpx::future::get). Concurrent access to the same shared state may conflict otherwise: for example multiple callers of hpx::shared_future::get must either all be read-only or provide external synchronization.

Public Functions

promise() = default

template<typename **Allocator**>
inline **promise**(*std*::allocator_arg_t, *Allocator* const &a)

promise(*promise* &&other) noexcept = default

promise(*promise* const &other) = delete

~promise() = default

promise &**operator=**(*promise* &&other) noexcept = default

promise &**operator=**(*promise* const &other) = delete

inline void **swap**(*promise* &other) noexcept

[712] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

inline void **set_value**(*R* const &r)

inline void **set_value**(*R* &&r)

template<typename ...**Ts**>
inline void **set_value**(*Ts*&&... ts)

Private Types

using **base_type** = detail::promise_base<*R*>

template<typename **R**>

class **promise**<*R*&> : public *hpx*::detail::promise_base<*R*&>

Public Functions

promise() = default

template<typename **Allocator**>
inline **promise**(*std*::allocator_arg_t, *Allocator* const &a)

promise(*promise* &&other) noexcept = default

promise(*promise* const &other) = delete

~promise() = default

promise &**operator=**(*promise* &&other) noexcept = default

promise &**operator=**(*promise* const &other) = delete

inline void **swap**(*promise* &other) noexcept

inline void **set_value**(*R* &r)

Private Types

using **base_type** = detail::promise_base<*R*&>

template<>

class **promise**<void> : public *hpx*::detail::promise_base<void>

Public Functions

promise() = default

template<typename **Allocator**>
inline **promise**(*std*::allocator_arg_t, *Allocator* const &a)

promise(*promise* &&other) noexcept = default

promise(*promise* const &other) noexcept = delete

~promise() = default

promise &**operator=**(*promise* &&other) noexcept = default

promise &**operator=**(*promise* const &other) noexcept = delete

inline void **swap**(*promise* &other) noexcept

inline void **set_value**()

Private Types

using **base_type** = detail::promise_base<void>

namespace **lcos**

namespace **local**

namespace **std**

Functions

template<typename **R**>
void **swap**(*hpx*::*promise*<*R*> &x, *hpx*::*promise*<*R*> &y) noexcept

template<typename R, typename Allocator> promise< R >, Allocator > : public true_type

io_service

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/io_service/io_service_pool.hpp

Defined in header hpx/io_service/io_service_pool.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

class **io_service_pool**

#include <io_service_pool.hpp> A pool of io_service objects.

Public Functions

HPX_NON_COPYABLE(*io_service_pool*)

explicit **io_service_pool**(*std*::size_t pool_size = 2, *threads*::*policies*::callback_notifier const &notifier = *threads*::*policies*::callback_notifier(), char const *pool_name = "", char const *name_postfix = "")

Construct the io_service pool.

Parameters

- **pool_size** – [in] The number of threads to run to serve incoming requests
- **notifier** – [in]
- **pool_name** – [in]
- **name_postfix** – [in]

explicit **io_service_pool**(*threads*::*policies*::callback_notifier const &notifier, char const *pool_name = "", char const *name_postfix = "")

Construct the io_service pool.

Parameters

- **notifier** – [in]
- **pool_name** – [in]
- **name_postfix** – [in]

~io_service_pool()

bool **run**(bool join_threads = true, *barrier* *startup = nullptr)

Run all io_service objects in the pool. If join_threads is true this will also wait for all threads to complete

bool **run**(*std*::size_t num_threads, bool join_threads = true, *barrier* *startup = nullptr)

Run all io_service objects in the pool. If join_threads is true this will also wait for all threads to complete

void **stop**()

Stop all io_service objects in the pool.

void **join**()

Join all io_service threads in the pool.

void **clear**()

Clear all internal data structures.

void **wait**()

Wait for all work to be done.

bool **stopped**()

asio::io_context &**get_io_service**(int index = -1)

Get an io_service to use.

std::thread &**get_os_thread_handle**(*std*::size_t thread_num)

access underlying thread handle

inline constexpr *std*::size_t **size**() const noexcept

Get number of threads associated with this I/O service.

void **thread_run**(*std*::size_t index, *barrier* *startup = nullptr) const

Activate the thread *index* for this thread pool.

inline constexpr char const ***get_name**() const noexcept

Return name of this pool.

void **init**(*std*::size_t pool_size)

Protected Functions

bool **run_locked**(*std*::size_t num_threads, bool join_threads, *barrier* *startup)

void **stop_locked**()

void **join_locked**()

void **clear_locked**()

void **wait_locked**()

Private Types

using **io_service_ptr** = *std*::unique_ptr<asio::io_context>

using **work_type** = *std*::unique_ptr<asio::io_context::work>

Private Members

std::mutex **mtx_**

std::vector<*io_service_ptr*> **io_services_**

The pool of io_services.

std::vector<*std*::thread> **threads_** *std*::vector<*work_type*> **work_**

The work that keeps the io_services running.

std::size_t **next_io_service_**

The next io_service to use for a connection.

bool **stopped_**

set to true if stopped

std::size_t **pool_size_**

initial number of OS threads to execute in this pool

threads::*policies*::callback_notifier const &**notifier_**

call this for each thread start/stop

char const ***pool_name_**

char const ***pool_name_postfix_**

bool **waiting_**

Set to true if waiting for work to finish.

std::unique_ptr<*barrier*> **wait_barrier_**

std::unique_ptr<*barrier*> **continue_barrier_**

Private Static Functions

static inline *work_type* **initialize_work**(asio::io_context &io_service)

lcos_local

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/lcos_local/trigger.hpp

Defined in header hpx/lcos_local/trigger.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **lcos**

namespace **local**

template<typename **Mutex** = *hpx*::*spinlock*>

struct **base_trigger**

Public Functions

inline **base_trigger**() noexcept

inline **base_trigger**(*base_trigger* &&rhs) noexcept

inline *base_trigger* &**operator=**(*base_trigger* &&rhs) noexcept

inline *hpx*::*future*<void> **get_future**(*std*::size_t *generation_value = nullptr, *error_code* &ec = *hpx*::*throws*)

get a future allowing to wait for the trigger to fire

inline bool **set**(*error_code* &ec = *throws*)

Trigger this object.

inline void **synchronize**(*std*::size_t generation_value, char const *function_name = "trigger::synchronize", *error_code* &ec = *throws*)

Wait for the generational counter to reach the requested stage.

inline *std*::size_t **next_generation**()

inline *std*::size_t **generation**() const

Protected Types

using **mutex_type** = *Mutex*

Protected Functions

inline bool **trigger_conditions**(*error_code* &ec = *throws*)

template<typename **Lock**>
inline void **synchronize**(*std*::size_t generation_value, *Lock* &l, char const *function_name = "trigger::synchronize", *error_code* &ec = *throws*)

Private Types

using **condition_list_type** = *hpx*::detail::intrusive_list<*condition_list_entry*>

Private Functions

inline bool **test_condition**(*std*::size_t generation_value) noexcept

Private Members

mutable *mutex_type* **mtx_**

hpx::*promise*<void> **promise_**

std::size_t **generation_**

condition_list_type **conditions_**

struct **condition_list_entry** : public conditional_trigger

Public Functions

condition_list_entry() = default

Public Members

condition_list_entry ***prev** = nullptr

condition_list_entry ***next** = nullptr

struct **manage_condition**

Public Functions

inline **manage_condition**(*base_trigger* &gate, *condition_list_entry* &cond) noexcept

inline **~manage_condition**()

template<typename **Condition**>
inline *hpx*::*future*<void> **get_future**(*Condition* &&func, *error_code* &ec = *hpx*::*throws*)

Public Members

base_trigger &**this_**

condition_list_entry &**e_**

struct **trigger** : public *hpx*::*lcos*::*local*::*base_trigger*<*hpx*::*no_mutex*>

Public Functions

trigger() = default

inline **trigger**(*trigger* &&rhs) noexcept

inline *trigger* &**operator=**(*trigger* &&rhs) noexcept

template<typename **Lock**>
inline void **synchronize**(*std*::size_t generation_value, *Lock* &l, char const *function_name = "trigger::synchronize", *error_code* &ec = *throws*)

Private Types

using **base_type** = *base_trigger*<*hpx*::*no_mutex*>

pack_traversal

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/pack_traversal/pack_traversal.hpp

Defined in header hpx/pack_traversal/pack_traversal.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

Functions

template<typename Mapper, typename...
T> < unspecified > map_pack (Mapper &&mapper, T &&... pack)

Maps the pack with the given mapper.

This function tries to visit all plain elements which may be wrapped in:
- homogeneous containers (`std::vector`, `std::list`)
- heterogeneous containers (`hpx::tuple`, `std::pair`, `std::array`) and re-assembles the pack with the result of the mapper. Mapping from one type to a different one is supported.

Elements that aren't accepted by the mapper are routed through and preserved through the hierarchy.

```
// Maps all integers to floats
map_pack([](int value) {
    return float(value);
},
1, hpx::make_tuple(2, std::vector<int>{3, 4}), 5);
```

Throws *std*`::exception` – like objects which are thrown by an invocation to the mapper.

Parameters
- **mapper** – A callable object, which accept an arbitrary type and maps it to another type or the same one.
- **pack** – An arbitrary variadic pack which may contain any type.

Returns The mapped element or in case the pack contains multiple elements, the pack is wrapped into a `hpx::tuple`.

hpx/pack_traversal/pack_traversal_async.hpp

Defined in header hpx/pack_traversal/pack_traversal_async.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

Functions

template<typename **Visitor**, typename ...**T**>
auto **traverse_pack_async**(*Visitor* &&visitor, *T*&&... pack) -> decltype(detail::apply_pack_transform_async(HPX_FORWARD(*Visitor*, *visitor*), HPX_FORWARD(*T*, *pack*)...))

Traverses the pack with the given visitor in an asynchronous way.

This function works in the same way as `traverse_pack`, however, we are able to suspend and continue the traversal at later time. Thus we require a visitor callable object which provides three `operator()` overloads as depicted by the code sample below:

```
    // The synchronous overload is called for each object, // it may
    return false to suspend the current control. // In that case the
    overload below is called. template <typename T> bool
    operator()(async_traverse_visit_tag, T&& element) {
        return true;
    }

    // The asynchronous overload this is called when the //
    synchronous overload returned false. // In addition to the
    current visited element the overload is // called with a
    continuation callable object which resumes the // traversal when
    it's called later. // The continuation next may be stored and
    called later or // dropped completely to abort the traversal
    early. template <typename T, typename N> void
    operator()(async_traverse_detach_tag, T&& element, N&& next) { }

    // The overload is called when the traversal was finished. // As
    argument the whole pack is passed over which we // traversed
    asynchronously. template <typename T> void
    operator()(async_traverse_complete_tag, T&& pack) { }
};
```

See `traverse_pack` for a detailed description about the traversal behavior and capabilities.

Parameters

- **visitor** – A visitor object which provides the three `operator()` overloads that were described above. Additionally the visitor must be compatible for referencing it from a `hpx::intrusive_ptr`. The visitor should must have a virtual destructor!
- **pack** – The arbitrary parameter pack which is traversed asynchronously. Nested objects inside containers and tuple like types are traversed recursively.

Returns A hpx::intrusive_ptr that references an instance of the given visitor object.

hpx::functional::unwrap, hpx::functional::unwrap_n, hpx::functional::unwrap_all, hpx::unwrap, hpx::unwrap_n, hpx::unwrap_all, hpx::unwrapping, hpx::unwrapping_n, hpx::unwrapping_all

Defined in header hpx/unwrap.hpp[713].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[713] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/unwrap.hpp

Functions

template<typename ...**Args**>
auto **unwrap**(*Args*&&... args) -> decltype(*util*::detail::unwrap_depth_impl<1U>(HPX_FORWARD(*Args*, *args*)...))

A helper function for retrieving the actual result of any hpx::future like type which is wrapped in an arbitrary way.

Unwraps the given pack of arguments, so that any hpx::future object is replaced by its future result type in the argument pack:

- hpx::future<int> -> int
- hpx::future<std::vector<float>> -> std::vector<float>
- std::vector<future<float>> -> std::vector<float>

The function is capable of unwrapping hpx::future like objects that are wrapped inside any container or tuple like type, see hpx::util::map_pack() for a detailed description about which surrounding types are supported. Non hpx::future like types are permitted as arguments and passed through.

```
hpx:unwrap(hpx::make_ready_future(0));

// Multiple arguments hpx::tuple<int, int> i2 =
    hpx:unwrap(hpx::make_ready_future(1),
               hpx::make_ready_future(2));
```

Note: This function unwraps the given arguments until the first traversed nested hpx::future which corresponds to an unwrapping depth of one. See hpx::unwrap_n() for a function which unwraps the given arguments to a particular depth or hpx::unwrap_all() that unwraps all future like objects recursively which are contained in the arguments.

Parameters **args** – the arguments that are unwrapped which may contain any arbitrary future or non future type.

Throws *std*::exception – like objects in case any of the given wrapped hpx::future objects were resolved through an exception. See hpx::future::get() for details.

Returns Depending on the count of arguments this function returns a hpx::tuple containing the unwrapped arguments if multiple arguments are given. In case the function is called with a single argument, the argument is unwrapped and returned.

template<*std*::size_t **Depth**, typename ...**Args**>
auto **unwrap_n**(*Args*&&... args) -> decltype(*util*::detail::unwrap_depth_impl<*Depth*>(HPX_FORWARD(*Args*, *args*)...))

An alterntive version of hpx::unwrap(), which unwraps the given arguments to a certain depth of hpx::future like objects.

See unwrap for a detailed description.

Template Parameters **Depth** – The count of hpx::future like objects which are unwrapped maximally.

template<typename ...**Args**>
auto **unwrap_all**(*Args*&&... args) ->
decltype(*util*::detail::unwrap_depth_impl<0U>(HPX_FORWARD(*Args*, *args*)...))

An alterntive version of hpx::unwrap(), which unwraps the given arguments recursively so that all contained hpx::future like objects are replaced by their actual value.

See hpx::unwrap() for a detailed description.

template<typename **T**>
auto **unwrapping**(*T* &&callable) ->
decltype(*util*::detail::functional_unwrap_depth_impl<1U>(HPX_FORWARD(*T*, *callable*)))

Returns a callable object which unwraps its arguments upon invocation using the hpx::unwrap() function and then passes the result to the given callable object.

```
    return left + right;
});

int i1 = callable(hpx::make_ready_future(1),
                  hpx::make_ready_future(2));
```

See hpx::unwrap() for a detailed description.

Parameters **callable** – the callable object which which is called with the result of the corresponding unwrap function.

template<*std*::size_t **Depth**, typename **T**>
auto **unwrapping_n**(*T* &&callable) ->
decltype(*util*::detail::functional_unwrap_depth_impl<*Depth*>(HPX_FORWARD(*T*, *callable*)))

Returns a callable object which unwraps its arguments upon invocation using the hpx::unwrap_n() function and then passes the result to the given callable object.

See hpx::unwrapping() for a detailed description.

template<typename **T**>
auto **unwrapping_all**(*T* &&callable) ->
decltype(*util*::detail::functional_unwrap_depth_impl<0U>(HPX_FORWARD(*T*, *callable*)))

Returns a callable object which unwraps its arguments upon invocation using the hpx::unwrap_all() function and then passes the result to the given callable object.

See hpx::unwrapping() for a detailed description.

namespace **functional**

struct **unwrap**

#include <unwrap.hpp> A helper function object for functionally invoking `hpx::unwrap`. For more information please refer to its documentation.

struct **unwrap_all**

#include <unwrap.hpp> A helper function object for functionally invoking `hpx::unwrap_all`. For more information please refer to its documentation.

template<*std*::size_t **Depth**>

struct **unwrap_n**

#include <unwrap.hpp> A helper function object for functionally invoking `hpx::unwrap_n`. For more information please refer to its documentation.

preprocessor

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/preprocessor/cat.hpp

Defined in header hpx/preprocessor/cat.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_PP_CAT(A, B)

Concatenates the tokens `A` and `B` into a single token. Evaluates to `AB`

Parameters

- **A** – First token
- **B** – Second token

hpx/preprocessor/expand.hpp

Defined in header hpx/preprocessor/expand.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_PP_EXPAND(X)

The HPX_PP_EXPAND macro performs a double macro-expansion on its argument.

This macro can be used to produce a delayed preprocessor expansion.

Example:

```
#define MACRO(a, b, c) (a)(b)(c)
#define ARGS() (1, 2, 3)

HPX_PP_EXPAND(MACRO ARGS()) // expands to (1)(2)(3)
```

Parameters

- **X** – Token to be expanded twice

hpx/preprocessor/nargs.hpp

Defined in header hpx/preprocessor/nargs.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_PP_NARGS(...)

Expands to the number of arguments passed in

Example Usage:

```
HPX_PP_NARGS(hpx, pp, nargs)
HPX_PP_NARGS(hpx, pp)
HPX_PP_NARGS(hpx)
```

Expands to:

```
3
2
1
```

Parameters

- **...** – The variadic number of arguments

hpx/preprocessor/stringize.hpp

Defined in header hpx/preprocessor/stringize.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_PP_STRINGIZE(X)

The *HPX_PP_STRINGIZE* macro stringizes its argument after it has been expanded.

The passed argument `X` will expand to `"X"`. Note that the stringizing operator (#) prevents arguments from expanding. This macro circumvents this shortcoming.

Parameters

- **X** – The text to be converted to a string literal

hpx/preprocessor/strip_parens.hpp

Defined in header hpx/preprocessor/strip_parens.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_PP_STRIP_PARENS(X)

For any symbol `X`, this macro returns the same symbol from which potential outer parens have been removed. If no outer parens are found, this macros evaluates to `X` itself without error.

The original implementation of this macro is from Steven Watanbe as shown in http://boost.2283326.n4.nabble.com/preprocessor-removing-parentheses-td2591973.html#a2591976

```
HPX_PP_STRIP_PARENS(no_parens)
HPX_PP_STRIP_PARENS((with_parens))
```

Example Usage:

This produces the following output

```
no_parens
with_parens
```

Parameters

- **X** – Symbol to strip parens from

resiliency

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/resiliency/replay_executor.hpp

Defined in header hpx/resiliency/replay_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **BaseExecutor**, typename **Validator**>

struct **is_two_way_executor**<*hpx*::*resiliency*::*experimental*::*replay_executor*<*BaseExecutor*, *Validator*>> : public true_type

template<typename **BaseExecutor**, typename **Validator**>

struct **is_bulk_two_way_executor**<*hpx*::*resiliency*::*experimental*::*replay_executor*<*BaseExecutor*, *Validator*>> : public true_type

namespace **hpx**

namespace **parallel**

namespace **execution**

template<typename BaseExecutor, typename Validator> replay_executor< BaseExecutor, Validator > > : public true_type

template<typename BaseExecutor, typename Validator> replay_executor< BaseExecutor, Validator > > : public true_type

namespace **resiliency**

namespace **experimental**

Functions

template<typename **Tag**, typename **BaseExecutor**, typename **Validate**, typename **Property**>
auto **tag_invoke**(*Tag* tag, *replay_executor*<*BaseExecutor*, *Validate*> const &exec, *Property* &&prop) -> decltype(*replay_executor*<*BaseExecutor*, *Validate*>(*std*::declval<*Tag*>()(*std*::declval<*BaseExecutor*>(), *std*::declval<*Property*>()), *std*::declval<*std*::size_t>(), *std*::declval<*Validate*>()))

template<typename **Tag**, typename **BaseExecutor**, typename **Validate**>
auto **tag_invoke**(*Tag* tag, *replay_executor*<*BaseExecutor*, *Validate*> const &exec) -> decltype(*std*::declval<*Tag*>()(*std*::declval<*BaseExecutor*>()))

template<typename **BaseExecutor**, typename **Validate**>
replay_executor<*BaseExecutor*, *std*::decay_t<*Validate*>> **make_replay_executor**(*BaseExecutor* &exec, *std*::size_t n, *Validate* &&validate)

template<typename **BaseExecutor**>
replay_executor<*BaseExecutor*, detail::replay_validator> **make_replay_executor**(*BaseExecutor* &exec, *std*::size_t n)

template<typename **BaseExecutor**, typename **Validate**>

class **replay_executor**

Public Types

using **execution_category** = *hpx*::*traits*::executor_execution_category_t<*BaseExecutor*>

using **executor_parameters_type** = *hpx*::*traits*::executor_parameters_type_t<*BaseExecutor*>

template<typename **Result**>

using **future_type** = *hpx*::*traits*::executor_future_t<*BaseExecutor*, *Result*>

Public Functions

template<typename **BaseExecutor_**, typename **F**>
inline explicit **replay_executor**(*BaseExecutor_* &&exec, *std*::size_t n, *F* &&f)

inline bool **operator==**(*replay_executor* const &rhs) const noexcept

inline bool **operator!=**(*replay_executor* const &rhs) const noexcept

inline constexpr *replay_executor* const &**context**() const noexcept

inline *BaseExecutor* const &**get_executor**() const

inline *std*::size_t **get_replay_count**() const

inline *Validate* const &**get_validator**() const

Public Static Attributes

static constexpr int **num_spread** = 4

static constexpr int **num_tasks** = 128

Private Functions

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*async_execute_t*, *replay_executor* const &exec, *F* &&f, *Ts*&&... ts)

template<typename **F**, typename **S**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_async_execute_t*, *replay_executor* const &exec, *F* &&f, *S* const &shape, *Ts*&&... ts)

Private Members

BaseExecutor **exec_**

std::size_t **replay_count_**

Validate **validator_**

hpx/resiliency/replicate_executor.hpp

Defined in header hpx/resiliency/replicate_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **BaseExecutor**, typename **Voter**, typename **Validator**>

struct **is_two_way_executor**<*hpx*::*resiliency*::*experimental*::*replicate_executor*<*BaseExecutor*, *Voter*, *Validator*>> : public true_type

template<typename **BaseExecutor**, typename **Voter**, typename **Validator**>

struct **is_bulk_two_way_executor**<*hpx*::*resiliency*::*experimental*::*replicate_executor*<*BaseExecutor*, *Voter*, *Validator*>> : public true_type

namespace **hpx**

namespace **parallel**

namespace **execution**

template<typename BaseExecutor, typename Voter, typename Validator> replicate_executor< BaseExecutor, Voter, Validator > > : public true_type

template<typename BaseExecutor, typename Voter, typename Validator> replicate_executor< BaseExecutor, Voter, Validator > > : public true_type

namespace **resiliency**

namespace **experimental**

Functions

template<typename **Tag**, typename **BaseExecutor**, typename **Vote**, typename **Validate**, typename **Property**>
auto **tag_invoke**(*Tag* tag, *replicate_executor*<*BaseExecutor*, *Vote*, *Validate*> const &exec, *Property* &&prop) -> decltype(*replicate_executor*<*BaseExecutor*, *Vote*, *Validate*>(*std*::declval<*Tag*>()(*std*::declval<*BaseExecutor*>(), *std*::declval<*Property*>()), *std*::declval<*std*::size_t>(), *std*::declval<*Vote*>(), *std*::declval<*Validate*>()))

template<typename **Tag**, typename **BaseExecutor**, typename **Vote**, typename **Validate**>
auto **tag_invoke**(*Tag* tag, *replicate_executor*<*BaseExecutor*, *Vote*, *Validate*> const &exec) -> decltype(*std*::declval<*Tag*>()(*std*::declval<*BaseExecutor*>()))

template<typename **BaseExecutor**, typename **Voter**, typename **Validate**>
replicate_executor<*BaseExecutor*, *std*::decay_t<*Voter*>, *std*::decay_t<*Validate*>> **make_replicate_executor**(*Base*
&exe
std::s
n,
Voter
&&v
Validate
&&va
idate)

template<typename **BaseExecutor**, typename **Validate**>
replicate_executor<*BaseExecutor*, detail::replicate_voter, *std*::decay_t<*Validate*>> **make_replicate_executor**(*Bas*
&ex
std:
n,
Validate
&&
idate

template<typename **BaseExecutor**>
replicate_executor<*BaseExecutor*, detail::replicate_voter, detail::replicate_validator> **make_replicate_executor**(*B*
&
st
n)

template<typename **BaseExecutor**, typename **Vote**, typename **Validate**>

class **replicate_executor**

Public Types

using **execution_category** = *hpx*::*traits*::executor_execution_category_t<*BaseExecutor*>

using **executor_parameters_type** = *hpx*::*traits*::executor_parameters_type_t<*BaseExecutor*>

template<typename **Result**>

using **future_type** = *hpx*::*traits*::executor_future_t<*BaseExecutor*, *Result*>

Public Functions

template<typename **BaseExecutor_**, typename **V**, typename **F**>
inline explicit **replicate_executor**(*BaseExecutor_* &&exec, *std*::size_t n, *V* &&v, *F* &&f)

inline bool **operator==**(*replicate_executor* const &rhs) const noexcept

inline bool **operator!=**(*replicate_executor* const &rhs) const noexcept

inline constexpr *replicate_executor* const &**context**() const noexcept

inline *BaseExecutor* const &**get_executor**() const

inline *std*::size_t **get_replicate_count**() const

inline *Vote* const &**get_voter**() const

inline *Validate* const &**get_validator**() const

Public Static Attributes

static constexpr int **num_spread** = 4

static constexpr int **num_tasks** = 128

Private Functions

template<typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*async_execute_t*, *replicate_executor* const &exec, *F* &&f, *Ts*&&... ts)

template<typename **F**, typename **S**, typename ...**Ts**>
inline decltype(auto) friend **tag_invoke**(*hpx*::*parallel*::*execution*::*bulk_async_execute_t*, *replicate_executor* const &exec, *F* &&f, *S* const &shape, *Ts*&&... ts)

Private Members

BaseExecutor **exec_**

std::size_t **replicate_count_**

Vote **voter_**

Validate **validator_**

runtime_configuration

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/runtime_configuration/component_commandline_base.hpp

Defined in header hpx/runtime_configuration/component_commandline_base.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_COMMANDLINE_REGISTRY(RegistryType, componentname)

The macro *HPX_REGISTER_COMMANDLINE_REGISTRY* is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the components.

HPX_REGISTER_COMMANDLINE_REGISTRY_DYNAMIC(RegistryType, componentname)

HPX_REGISTER_COMMANDLINE_OPTIONS()

The macro *HPX_REGISTER_COMMANDLINE_OPTIONS* is used to define the required Hpx.Plugin entry point for the command line option registry. This macro has to be used in not more than one compilation unit of a component module.

HPX_REGISTER_COMMANDLINE_OPTIONS_DYNAMIC()

namespace **hpx**

namespace **components**

struct **component_commandline_base**

#include <component_commandline_base.hpp> The component_commandline_base has to be used as a base class for all component command-line line handling registries.

Public Functions

virtual **~component_commandline_base**() = default

virtual *hpx*::program_options::options_description **add_commandline_options**() = 0

Return any additional command line options valid for this component.

Note: This function will be executed by the runtime system during system startup.

Returns The module is expected to fill a options_description object with any additional command line options this component will handle.

hpx/runtime_configuration/component_registry_base.hpp

Defined in header hpx/runtime_configuration/component_registry_base.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_COMPONENT_REGISTRY(RegistryType, componentname)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the components.

HPX_REGISTER_COMPONENT_REGISTRY_DYNAMIC(RegistryType, componentname)

HPX_REGISTER_REGISTRY_MODULE()

This macro is used to define the required Hpx.Plugin entry points. This macro has to be used in exactly one compilation unit of a component module.

HPX_REGISTER_REGISTRY_MODULE_DYNAMIC()

namespace **hpx**

namespace **components**

struct **component_registry_base**

#include <component_registry_base.hpp> The component_registry_base has to be used as a base class for all component registries.

Public Functions

virtual **~component_registry_base**() = default

virtual bool **get_component_info**(*std*::vector<*std*::string> &fillini, *std*::string const &filepath, bool is_static = false) = 0

Return the ini-information for all contained components.

Parameters

- **fillini** – [in, out] The module is expected to fill this vector with the ini-information (one line per vector element) for all components implemented in this module.

- **filepath** – [in]
- **is_static** – [in]

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

virtual void **register_component_type**() = 0

Register the component type represented by this component.

hpx/runtime_configuration/plugin_registry_base.hpp

Defined in header hpx/runtime_configuration/plugin_registry_base.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_PLUGIN_BASE_REGISTRY(PluginType, name)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the components.

HPX_REGISTER_PLUGIN_REGISTRY_MODULE()

This macro is used to define the required Hpx.Plugin entry points. This macro has to be used in exactly one compilation unit of a component module.

HPX_REGISTER_PLUGIN_REGISTRY_MODULE_DYNAMIC()

namespace **hpx**

namespace **plugins**

struct **plugin_registry_base**

#include <plugin_registry_base.hpp> The plugin_registry_base has to be used as a base class for all plugin registries.

Public Functions

virtual **~plugin_registry_base**() = default

virtual bool **get_plugin_info**(*std*::vector<*std*::string> &fillini) = 0

Return the configuration information for any plugin implemented by this module

Parameters fillini – [in, out] The module is expected to fill this vector with the ini-information (one line per vector element) for all plugins implemented in this module.

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

inline virtual void **init**(int*, char***, *util*::runtime_configuration&)

hpx::runtime_mode

Defined in header hpx/init.hpp[714].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Enums

enum class **runtime_mode**

A HPX runtime can be executed in two different modes: console mode and worker mode.

Values:

enumerator **invalid**

enumerator **console**

The runtime is the console locality.

enumerator **worker**

The runtime is a worker locality.

enumerator **connect**

The runtime is a worker locality connecting late

enumerator **local**

The runtime is fully local.

enumerator **default_**

The runtime mode will be determined based on the command line arguments

enumerator **last**

Functions

char const ***get_runtime_mode_name**(*runtime_mode* state) noexcept

Get the readable string representing the name of the given runtime_mode constant.

runtime_mode **get_runtime_mode_from_name**(*std*::string const &mode)

Returns the internal representation (runtime_mode constant) from the readable string representing the name.

This represents the internal representation from the readable string representing the name.

Parameters **mode** – this represents the runtime mode

[714] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/init_runtime/include/hpx/init.hpp

runtime_local

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/runtime_local/component_startup_shutdown_base.hpp

Defined in header hpx/runtime_local/component_startup_shutdown_base.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_STARTUP_SHUTDOWN_REGISTRY(RegistryType, componentname)

This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the components.

HPX_REGISTER_STARTUP_SHUTDOWN_REGISTRY_DYNAMIC(RegistryType, componentname)

HPX_REGISTER_STARTUP_SHUTDOWN_FUNCTIONS()

This macro is used to define the required Hpx.Plugin entry point for the startup/shutdown registry. This macro has to be used in not more than one compilation unit of a component module.

HPX_REGISTER_STARTUP_SHUTDOWN_FUNCTIONS_DYNAMIC()

namespace **hpx**

namespace **components**

struct **component_startup_shutdown_base**

#include <component_startup_shutdown_base.hpp> The component_startup_shutdown_base has to be used as a base class for all component startup/shutdown registries.

Public Functions

virtual **~component_startup_shutdown_base**() = default

virtual bool **get_startup_function**(*startup_function_type* &startup, bool &pre_startup) = 0

Return any startup function for this component.

Parameters

- **startup** – [in, out] The module is expected to fill this function object with a reference to a startup function. This function will be executed by the runtime system during system startup.
- **pre_startup** – [in, out] Will be set to true if the returned startup function is executed during the first round of calls.

Returns Returns *true* if the parameter *startup* has been successfully initialized with the startup function.

virtual bool **get_shutdown_function**(*shutdown_function_type* &shutdown, bool &pre_shutdown) = 0

Return any startup function for this component.

Parameters

- **shutdown** – [in, out] The module is expected to fill this function object with a reference to a startup function. This function will be executed by the runtime system during system startup.
- **pre_shutdown** – [in, out] Will be set to true if the returned shutdown function is executed during the first round of calls.

Returns Returns *true* if the parameter *shutdown* has been successfully initialized with the shutdown function.

hpx/runtime_local/custom_exception_info.hpp

Defined in header hpx/runtime_local/custom_exception_info.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::string **diagnostic_information**(exception_info const &xi)

Extract the diagnostic information embedded in the given exception and return a string holding a formatted message.

The function *hpx::diagnostic_information* can be used to extract all diagnostic information stored in the given exception instance as a formatted string. This simplifies debug output as it composes the diagnostics into one, easy to use function call. This includes the name of the source file and line number, the sequence number of the OS-thread and the HPX-thread id, the locality id and the stack backtrace of the point where the original exception was thrown.

See also:

hpx::get_error_locality_id(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters **xi** – The parameter `e` will be inspected for all diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if any of the required allocation operations fail)

Returns The formatted string holding all the available diagnostic information stored in the given exception instance.

std::uint32_t **get_error_locality_id**(*hpx*::exception_info const &xi) noexcept

Return the locality id where the exception was thrown.

The function *hpx::get_error_locality_id* can be used to extract the diagnostic information element representing the locality id as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters `xi` – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `nothing` –

Returns The locality id of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return *hpx::naming::invalid_locality_id*.

std::string **`get_error_host_name`**(*hpx*::exception_info const &xi)

Return the hostname of the locality where the exception was thrown.

The function *hpx::get_error_host_name* can be used to extract the diagnostic information element representing the host name as stored in the given exception instance.

See also:

hpx::diagnostic_information() *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error()* *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters `xi` – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns The hostname of the locality where the exception was thrown. If the exception instance does not hold this information, the function will return and empty string.

std::int64_t **`get_error_process_id`**(*hpx*::exception_info const &xi) noexcept

Return the (operating system) process id of the locality where the exception was thrown.

The function *hpx::get_error_process_id* can be used to extract the diagnostic information element representing the process id as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters xi – The parameter e will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws nothing –

Returns The process id of the OS-process which threw the exception If the exception instance does not hold this information, the function will return 0.

std::string **get_error_env**(*hpx*::exception_info const &xi)

Return the environment of the OS-process at the point the exception was thrown.

The function *hpx::get_error_env* can be used to extract the diagnostic information element representing the environment of the OS-process collected at the point the exception was thrown.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error()*, *hpx::get_error_backtrace()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters xi – The parameter e will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws std::bad_alloc – (if one of the required allocations fails)

Returns The environment from the point the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

std::string **get_error_backtrace**(*hpx*::exception_info const &xi)

Return the stack backtrace from the point the exception was thrown.

The function *hpx::get_error_backtrace* can be used to extract the diagnostic information element representing the stack backtrace collected at the point the exception was thrown.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error()*, *hpx::get_error_env()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters xi – The parameter e will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws std::bad_alloc – (if one of the required allocations fails)

Returns The stack back trace from the point the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

std::size_t **get_error_os_thread**(*hpx*::exception_info const &xi) noexcept

Return the sequence number of the OS-thread used to execute HPX-threads from which the exception was thrown.

The function *hpx::get_error_os_thread* can be used to extract the diagnostic information element representing the sequence number of the OS-thread as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_thread_id()*, *hpx::get_error_thread_description()*, *hpx::get_error()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters **xi** – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `nothing` –

Returns The sequence number of the OS-thread used to execute the HPX-thread from which the exception was thrown. If the exception instance does not hold this information, the function will return std::size(-1).

std::size_t **get_error_thread_id**(*hpx*::exception_info const &xi) noexcept

Return the unique thread id of the HPX-thread from which the exception was thrown.

The function *hpx::get_error_thread_id* can be used to extract the diagnostic information element representing the HPX-thread id as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()* *hpx::get_error_thread_description()*, *hpx::get_error()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_what()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters **xi** – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `nothing` –

Returns The unique thread id of the HPX-thread from which the exception was thrown. If the exception instance does not hold this information, the function will return std::size_t(0).

std::string **`get_error_thread_description`**(*hpx*::exception_info const &xi)

Return any additionally available thread description of the HPX-thread from which the exception was thrown.

The function *hpx::get_error_thread_description* can be used to extract the diagnostic information element representing the additional thread description as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error()*, *hpx::get_error_state()*, *hpx::get_error_what()*, *hpx::get_error_config()*

Parameters **`xi`** – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns Any additionally available thread description of the HPX-thread from which the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

std::string **`get_error_config`**(*hpx*::exception_info const &xi)

Return the HPX configuration information point from which the exception was thrown.

The function *hpx::get_error_config* can be used to extract the HPX configuration information element representing the full HPX configuration information as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error()*, *hpx::get_error_state()* *hpx::get_error_what()*, *hpx::get_error_thread_description()*

Parameters **`xi`** – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns Any additionally available HPX configuration information the point from which the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

std::string **`get_error_state`**(*hpx*::exception_info const &xi)

Return the HPX runtime state information at which the exception was thrown.

The function *hpx::get_error_state* can be used to extract the HPX runtime state information element representing the state the runtime system is currently in as stored in the given exception instance.

See also:

hpx::diagnostic_information(), *hpx::get_error_host_name()*, *hpx::get_error_process_id()*, *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, *hpx::get_error_os_thread()*, *hpx::get_error_thread_id()*, *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error()*, *hpx::get_error_what()*, *hpx::get_error_thread_description()*

Parameters `xi` – The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Throws `std::bad_alloc` – (if one of the required allocations fails)

Returns The point runtime state at the point at which the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

hpx::get_locality_id

Defined in header hpx/runtime.hpp[715].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::uint32_t **get_locality_id**(*error_code* &ec = *throws*)

Return the number of the locality this function is being called from.

This function returns the id of the current locality.

Note: The returned value is zero based and its maximum value is smaller than the overall number of localities the current application is running on (as returned by get_num_localities()).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise, it throws an instance of hpx::exception.

Note: This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters `ec` – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

[715] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

hpx::get_locality_name

Defined in header hpx/runtime.hpp[716].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::string **get_locality_name**()

Return the name of the locality this function is called on.

This function returns the name for the locality on which this function is called.

See also:

future<std::string> get_locality_name(hpx::id_type const& id)

Returns This function returns the name for the locality on which the function is called. The name is retrieved from the underlying networking layer and may be different for different parcelports.

hpx::get_initial_num_localities, hpx::get_num_localities

Defined in header hpx/runtime.hpp[717].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::uint32_t **get_initial_num_localities**()

Return the number of localities which were registered at startup for the running application.

The function *get_initial_num_localities* returns the number of localities which were connected to the console at application startup.

See also:

hpx::find_all_localities, *hpx::get_num_localities*

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise, it throws an instance of hpx::exception.

[716] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp
[717] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

hpx::*future*<*std*::uint32_t> **get_num_localities**()

Asynchronously return the number of localities which are currently registered for the running application.

The function *get_num_localities* asynchronously returns the number of localities currently connected to the console. The returned future represents the actual result.

See also:

hpx::find_all_localities, *hpx::get_num_localities*

Note: This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

std::uint32_t **get_num_localities**(*launch*::sync_policy, *error_code* &ec = *throws*)

Return the number of localities which are currently registered for the running application.

The function *get_num_localities* returns the number of localities currently connected to the console.

See also:

hpx::find_all_localities, *hpx::get_num_localities*

Note: This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise, it throws an instance of hpx::exception.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

hpx/runtime_local/get_os_thread_count.hpp

Defined in header hpx/runtime_local/get_os_thread_count.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::size_t **get_os_thread_count**()

Return the number of OS-threads running in the runtime instance the current HPX-thread is associated with.

std::size_t **get_os_thread_count**(*threads*::executor const &exec)

Return the number of worker OS- threads used by the given executor to execute HPX threads.

This function returns the number of cores used to execute HPX threads for the given executor. If the function is called while no HPX runtime system is active, it will return zero. If the executor is not valid, this function will fall back to retrieving the number of OS threads used by HPX.

Parameters **exec** – [in] The executor to be used.

namespace **threads**

hpx::get_thread_name

Defined in header hpx/runtime.hpp[718].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::string **get_thread_name**()

Return the name of the calling thread.

This function returns the name of the calling thread. This name uniquely identifies the thread in the context of HPX. If the function is called while no HPX runtime system is active, the result will be “<unknown>”.

hpx/runtime_local/get_worker_thread_num.hpp

Defined in header hpx/runtime_local/get_worker_thread_num.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/runtime_local/report_error.hpp

Defined in header hpx/runtime_local/report_error.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[718] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

Functions

void **report_error**(*std*::size_t num_thread, *std*::exception_ptr const &e)
The function report_error reports the given exception to the console.

void **report_error**(*std*::exception_ptr const &e)
The function report_error reports the given exception to the console.

hpx/runtime_local/runtime_local.hpp

Defined in header hpx/runtime_local/runtime_local.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

void **set_error_handlers**(*hpx*::*util*::runtime_configuration const &cfg)

class **runtime**

Public Types

using **notification_policy_type** = *threads*::*policies*::callback_notifier
Generate a new notification policy instance for the given thread name prefix

using **hpx_main_function_type** = int()
The *hpx_main_function_type* is the default function type usable as the main HPX thread function.

using **hpx_errorsink_function_type** = void(*std*::uint32_t, *std*::string const&)

Public Functions

virtual *notification_policy_type* **get_notification_policy**(char const *prefix, runtime_local::os_thread_type type)

state **get_state**() const

void **set_state**(state s)

explicit **runtime**(*hpx*::*util*::runtime_configuration rtcfg, bool initialize)
Construct a new HPX runtime instance.

virtual **~runtime**()
The destructor makes sure all HPX runtime services are properly shut down before exiting.

void **on_exit**(*hpx*::*function*<void()> const &f)
Manage list of functions to call on exit.

void **starting**()

Manage runtime 'stopped' state.

void **stopping**()

Call all registered on_exit functions.

bool **stopped**() const

This accessor returns whether the runtime instance has been stopped.

hpx::*util*::runtime_configuration &**get_config**()

access configuration information

hpx::*util*::runtime_configuration const &**get_config**() const

std::size_t **get_instance_number**() const

util::thread_mapper &**get_thread_mapper**() const

Return a reference to the internal PAPI thread manager.

threads::*topology* const &**get_topology**() const

virtual int **run**(*hpx*::*function*<*hpx_main_function_type*> const &func)

Run the HPX runtime system, use the given function for the main *thread* and block waiting for all threads to finish.

Note: The parameter *func* is optional. If no function is supplied, the runtime system will simply wait for the shutdown action without explicitly executing any main thread.

Parameters `func` – [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the typedef *hpx_main_function_type*. This parameter is optional and defaults to none main thread function, in which case all threads have to be scheduled explicitly.

Returns This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

virtual int **run**()

Run the HPX runtime system, initially use the given number of (OS) threads in the thread-manager and block waiting for all threads to finish.

Returns This function will always return 0 (zero).

virtual void **rethrow_exception**()

Rethrow any stored exception (to be called after *stop()*)

virtual int **start**(*hpx*::*function*<*hpx_main_function_type*> const &func, bool blocking = false)

Start the runtime system.

Parameters

- `func` – [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the typedef *hpx_main_function_type*.
- `blocking` – [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function runtime::start will call runtime::wait internally.

Returns If a blocking is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter `func`. Otherwise, it will return zero.

virtual int **start**(bool blocking = false)

Start the runtime system.

Parameters **blocking** – [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function runtime::start will call runtime::wait internally .

Returns If a blocking is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter `func`. Otherwise, it will return zero.

virtual int **wait**()

Wait for the shutdown action to be executed.

Returns This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

virtual void **stop**(bool blocking = true)

Initiate termination of the runtime system.

Parameters **blocking** – [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

virtual int **suspend**()

Suspend the runtime system.

virtual int **resume**()

Resume the runtime system.

virtual int **finalize**(double)

virtual bool **is_networking_enabled**()

Return true if networking is enabled.

virtual *hpx*::*threads*::*threadmanager* &**get_thread_manager**()

Allow access to the thread manager instance used by the HPX runtime.

virtual *std*::string **here**() const

Returns a string of the locality endpoints (usable in debug output)

virtual bool **report_error**(*std*::size_t num_thread, *std*::exception_ptr const &e, bool terminate_all = true)

Report a non-recoverable error to the runtime system.

Parameters

- **num_thread** – [in] The number of the operating system thread the error has been detected in.
- **e** – [in] This is an instance encapsulating an exception which lead to this function call.
- **terminate_all** – [in] signal whether all localities should be terminated

virtual bool **report_error**(*std*::exception_ptr const &e, bool terminate_all = true)

Report a non-recoverable error to the runtime system.

Note: This function will retrieve the number of the current shepherd thread and forward to the report_error function above.

Parameters

- **e** – [in] This is an instance encapsulating an exception which lead to this function call.

- **terminate_all** – [in] signal whether all localities should be terminated

virtual void **add_pre_startup_function**(*startup_function_type* f)

Add a function to be executed inside a HPX thread before hpx_main but guaranteed to be executed before any startup function registered with *add_startup_function*.

Note: The difference to a startup function is that all pre-startup functions will be (system-wide) executed before any startup function.

Parameters **f** – The function 'f' will be called from inside a HPX thread before hpx_main is executed. This is very useful to set up the runtime environment of the application (install performance counters, etc.)

virtual void **add_startup_function**(*startup_function_type* f)

Add a function to be executed inside a HPX thread before hpx_main

Parameters **f** – The function 'f' will be called from inside a HPX thread before hpx_main is executed. This is very useful to set up the runtime environment of the application (install performance counters, etc.)

virtual void **add_pre_shutdown_function**(*shutdown_function_type* f)

Add a function to be executed inside a HPX thread during hpx::finalize, but guaranteed before any of the shutdown functions is executed.

Note: The difference to a shutdown function is that all pre-shutdown functions will be (system-wide) executed before any shutdown function.

Parameters **f** – The function 'f' will be called from inside a HPX thread while hpx::finalize is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

virtual void **add_shutdown_function**(*shutdown_function_type* f)

Add a function to be executed inside a HPX thread during hpx::finalize

Parameters **f** – The function 'f' will be called from inside a HPX thread while hpx::finalize is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

virtual *hpx*::*util*::*io_service_pool* ***get_thread_pool**(char const *name)

Access one of the internal thread pools (io_service instances) HPX is using to perform specific tasks. The three possible values for the argument `name` are "main_pool", "io_pool", "parcel_pool", and "timer_pool". For any other argument value the function will return zero.

virtual bool **register_thread**(char const *name, *std*::size_t num = 0, bool service_thread = true, *error_code* &ec = *throws*)

Register an external OS-thread with HPX.

This function should be called from any OS-thread which is external to HPX (not created by HPX), but which needs to access HPX functionality, such as setting a value on a promise or similar.

'main', 'io', 'timer', 'parcel', 'worker'

Note: The function will compose a thread name of the form '<name>-thread#<num>' which is used to register the thread. It is the user's responsibility to ensure that each (composed) thread name is unique. HPX internally uses the following names for the threads it creates, do not reuse those:

Note: This function should be called for each thread exactly once. It will fail if it is called more than once.

Parameters

- **name** – [in] The name to use for thread registration.
- **num** – [in] The sequence number to use for thread registration. The default for this parameter is zero.
- **service_thread** – [in] The thread should be registered as a service thread. The default for this parameter is 'true'. Any service threads will be pinned to cores not currently used by any of the HPX worker threads.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will return whether the requested operation succeeded or not.

virtual bool **unregister_thread**()

Unregister an external OS-thread with HPX.

This function will unregister any external OS-thread from HPX.

Note: This function should be called for each thread exactly once. It will fail if it is called more than once. It will fail as well if the thread has not been registered before (see *register_thread*).

Returns This function will return whether the requested operation succeeded or not.

virtual runtime_local::os_thread_data **get_os_thread_data**(*std*::string const &label) const

Access data for a given OS thread that was previously registered by *register_thread*.

virtual bool **enumerate_os_threads**(*hpx*::*function*<bool(runtime_local::os_thread_data const&)> const &f) const

Enumerate all OS threads that have registered with the runtime.

notification_policy_type::on_startstop_type **on_start_func**() const

notification_policy_type::on_startstop_type **on_stop_func**() const

notification_policy_type::on_error_type **on_error_func**() const

notification_policy_type::on_startstop_type **on_start_func**(*notification_policy_type*::on_startstop_type&&)

notification_policy_type::on_startstop_type **on_stop_func**(*notification_policy_type*::on_startstop_type&&)

notification_policy_type::on_error_type **on_error_func**(*notification_policy_type*::on_error_type&&)

virtual *std*::uint32_t **get_locality_id**(*error_code* &ec) const

virtual *std*::size_t **get_num_worker_threads**() const

virtual *std*::uint32_t **get_num_localities**(*hpx*::*launch*::sync_policy, *error_code* &ec) const

virtual *std*::uint32_t **get_initial_num_localities**() const

virtual *hpx*::*future*<*std*::uint32_t> **get_num_localities**() const virtual *std*::string **get_locality_name**() const

virtual *std*::uint32_t **assign_cores**(*std*::string const&, *std*::uint32_t)

virtual *std*::uint32_t **assign_cores**()

inline *hpx*::program_options::options_description const &**get_app_options**() const

inline void **set_app_options**(*hpx*::program_options::options_description const &app_options)

Public Static Functions

static *std*::uint64_t **get_system_uptime**()

Return the system uptime measure on the thread executing this call.

Protected Types

using **on_exit_type** = *std*::vector<*hpx*::*function*<void()>>

Protected Functions

explicit **runtime**(*hpx*::*util*::runtime_configuration rtcfg)

void **set_notification_policies**(*notification_policy_type* &&notifier, *threads*::detail::network_background_callback_type const &network_background_callback)

void **init**()

Common initialization for different constructors.

void **init_global_data**()

threads::thread_result_type **run_helper**(*hpx*::*function*<*runtime*::*hpx_main_function_type*> const &func, int &result, bool call_startup_functions, void (*handle_print_bind)(*std*::size_t) = nullptr)

void **wait_helper**(*std*::mutex &mtx, *std*::condition_variable &cond, bool &running)

Protected Attributes

on_exit_type **on_exit_functions_**

mutable *std*::mutex **mtx_**

hpx::*util*::runtime_configuration **rtcfg_**

long **instance_number_**

std::unique_ptr<*util*::thread_mapper> **thread_support_**

threads::*topology* &**topology_**

std::atomic<state> **state_**

notification_policy_type::on_startstop_type **on_start_func_**

notification_policy_type::on_startstop_type **on_stop_func_**

notification_policy_type::on_error_type **on_error_func_**

int **result_**

std::exception_ptr **exception_**

notification_policy_type **main_pool_notifier_**

std::unique_ptr<*util*::*io_service_pool*> **main_pool_**

notification_policy_type **notifier_**

std::unique_ptr<*hpx*::*threads*::*threadmanager*> **thread_manager_**

Protected Static Functions

static void **deinit_global_data**()

Protected Static Attributes

static *std*::atomic<int> **instance_number_counter_**

Private Functions

void **stop_helper**(bool blocking, *std*::condition_variable &cond, *std*::mutex &mtx) const

Helper function to stop the runtime.

Parameters

- **blocking** – [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.
- **cond** –
- **mtx** –

void **deinit_tss_helper**(char const *context, *std*::size_t num) const

void **init_tss_ex**(char const *context, runtime_local::os_thread_type type, *std*::size_t local_thread_num, *std*::size_t global_thread_num, char const *pool_name, char const *postfix, bool service_thread, *error_code* &ec) const

void **init_tss_helper**(char const *context, runtime_local::os_thread_type type, *std*::size_t local_thread_num, *std*::size_t global_thread_num, char const *pool_name, char const *postfix, bool service_thread) const

void **notify_finalize**()

void **wait_finalize**()

void **call_startup_functions**(bool pre_startup)

Private Members

std::list<*startup_function_type*> **pre_startup_functions_**

std::list<*startup_function_type*> **startup_functions_**

std::list<*shutdown_function_type*> **pre_shutdown_functions_**

std::list<*shutdown_function_type*> **shutdown_functions_**

bool **stop_called_**

bool **stop_done_**

std::condition_variable **wait_condition_**

hpx::program_options::options_description **app_options_**

namespace **threads**

Functions

char const ***get_stack_size_name**(*std*::ptrdiff_t size)

Returns the stack size name.

Get the readable string representing the given stack size constant.

Parameters **size** – this represents the stack size

std::ptrdiff_t **get_default_stack_size**()

Returns the default stack size.

Get the default stack size in bytes.

std::ptrdiff_t **get_stack_size**(*thread_stacksize*)

Returns the stack size corresponding to the given stack size enumeration.

Get the stack size corresponding to the given stack size enumeration.

Parameters **size** – this represents the stack size

namespace **util**

Functions

bool **retrieve_commandline_arguments**(*hpx*::program_options::options_description const &app_options, *hpx*::program_options::variables_map &vm)

bool **retrieve_commandline_arguments**(*std*::string const &appname, *hpx*::program_options::variables_map &vm)

hpx::register_thread, hpx::unregister_thread, hpx::get_os_thread_data, hpx::enumerate_os_threads, hpx::get_runtime_instance_number, hpx::register_on_exit, hpx::is_starting, hpx::tolerate_node_faults, hpx::is_running, hpx::is_stopped, hpx::is_stopped_or_shutting_down, hpx::get_num_worker_threads, hpx::get_system_uptime

Defined in header hpx/runtime.hpp[719].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

bool **register_thread**(*runtime* *rt, char const *name, *error_code* &ec = *throws*)

Register the current kernel thread with HPX, this should be done once for each external OS-thread intended to invoke HPX functionality. Calling this function more than once will return false.

void **unregister_thread**(*runtime* *rt)

Unregister the thread from HPX, this should be done once in the end before the external thread exists.

runtime_local::os_thread_data **get_os_thread_data**(*std*::string const &label)

Access data for a given OS thread that was previously registered by *register_thread*. This function must be called from a thread that was previously registered with the runtime.

bool **enumerate_os_threads**(*hpx*::*function*<bool(os_thread_data const&)> const &f)

Enumerate all OS threads that have registered with the runtime.

std::size_t **get_runtime_instance_number**()

Return the runtime instance number associated with the runtime instance the current thread is running in.

bool **register_on_exit**(*hpx*::*function*<void()> const&)

Register a function to be called during system shutdown.

[719] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

bool **is_starting**()

Test whether the runtime system is currently being started.

This function returns whether the runtime system is currently being started or not, e.g. whether the current state of the runtime system is *hpx::state::startup*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

bool **tolerate_node_faults**()

Test if HPX runs in fault-tolerant mode.

This function returns whether the runtime system is running in fault-tolerant mode

bool **is_running**()

Test whether the runtime system is currently running.

This function returns whether the runtime system is currently running or not, e.g. whether the current state of the runtime system is *hpx::state::running*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

bool **is_stopped**()

Test whether the runtime system is currently stopped.

This function returns whether the runtime system is currently stopped or not, e.g. whether the current state of the runtime system is *hpx::state::stopped*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

bool **is_stopped_or_shutting_down**()

Test whether the runtime system is currently being shut down.

This function returns whether the runtime system is currently being shut down or not, e.g. whether the current state of the runtime system is *hpx::state::stopped* or *hpx::state::shutdown*

Note: This function needs to be executed on a HPX-thread. It will return false otherwise.

std::size_t **get_num_worker_threads**()

Return the number of worker OS- threads used to execute HPX threads.

This function returns the number of OS-threads used to execute HPX threads. If the function is called while no HPX runtime system is active, it will return zero.

std::uint64_t **get_system_uptime**()

Return the system uptime measure on the thread executing this call.

This function returns the system uptime measured in nanoseconds for the thread executing this call. If the function is called while no HPX runtime system is active, it will return zero.

namespace **threads**

hpx/runtime_local/service_executors.hpp

Defined in header hpx/runtime_local/service_executors.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Enums

enum class **service_executor_type** : *std*::uint8_t

Values:

enumerator **io_thread_pool**

Selects creating a service executor using the I/O pool of threads

enumerator **parcel_thread_pool**

Selects creating a service executor using the parcel pool of threads

enumerator **timer_thread_pool**

Selects creating a service executor using the timer pool of threads

enumerator **main_thread**

Selects creating a service executor using the main thread

struct **io_pool_executor** : public *service_executor*

Public Functions

io_pool_executor()

struct **main_pool_executor** : public *service_executor*

Public Functions

main_pool_executor()

struct **parcel_pool_executor** : public *service_executor*

Public Functions

explicit **parcel_pool_executor**(char const *name_suffix = "-tcp")

struct **service_executor** : public *service_executor*

Public Functions

explicit **service_executor**(*service_executor_type* t, char const *name_suffix = "")

struct **timer_pool_executor** : public *service_executor*

Public Functions

timer_pool_executor()

hpx::shutdown_function_type, hpx::register_pre_shutdown_function, hpx::register_shutdown_function

Defined in header hpx/runtime.hpp[720].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Typedefs

using **shutdown_function_type** = *hpx*::*move_only_function*<void()>

The type of the function which is registered to be executed as a shutdown or pre-shutdown function.

Functions

void **register_pre_shutdown_function**(*shutdown_function_type* f)

Add a function to be executed by a HPX thread during *hpx::finalize()* but guaranteed before any shutdown function is executed (system-wide)

Any of the functions registered with *register_pre_shutdown_function* are guaranteed to be executed by an HPX thread during the execution of *hpx::finalize()* before any of the registered shutdown functions are executed (see: hpx::register_shutdown_function()).

[720] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

See also:

hpx::register_shutdown_function()

Note: If this function is called while the pre-shutdown functions are being executed, or after that point, it will raise an invalid_status exception.

Parameters **f** – [in] The function to be registered to run by an HPX thread as a pre-shutdown function.

void **register_shutdown_function**(*shutdown_function_type* f)

Add a function to be executed by a HPX thread during *hpx::finalize()* but guaranteed after any pre-shutdown function is executed (system-wide)

Any of the functions registered with *register_shutdown_function* are guaranteed to be executed by an HPX thread during the execution of *hpx::finalize()* after any of the registered pre-shutdown functions are executed (see: hpx::register_pre_shutdown_function()).

See also:

hpx::register_pre_shutdown_function()

Note: If this function is called while the shutdown functions are being executed, or after that point, it will raise an invalid_status exception.

Parameters **f** – [in] The function to be registered to run by an HPX thread as a shutdown function.

hpx::startup_function_type, hpx::register_pre_startup_function, hpx::register_startup_function

Defined in header hpx/runtime.hpp[721].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Typedefs

using **startup_function_type** = *hpx*::*move_only_function*<void()>

The type of the function which is registered to be executed as a startup or pre-startup function.

[721] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

Functions

void **register_pre_startup_function**(*startup_function_type* f)

Add a function to be executed by a HPX thread before hpx_main but guaranteed before any startup function is executed (system-wide).

Any of the functions registered with *register_pre_startup_function* are guaranteed to be executed by an HPX thread before any of the registered startup functions are executed (see hpx::register_startup_function()).

This function is one of the few API functions which can be called before the runtime system has been fully initialized. It will automatically stage the provided startup function to the runtime system during its initialization (if necessary).

See also:

hpx::register_startup_function()

Note: If this function is called while the pre-startup functions are being executed or after that point, it will raise aninvalid_status exception.

Parameters **f** – [in] The function to be registered to run by an HPX thread as a pre-startup function.

void **register_startup_function**(*startup_function_type* f)

Add a function to be executed by a HPX thread before hpx_main but guaranteed after any pre-startup function is executed (system-wide).

Any of the functions registered with *register_startup_function* are guaranteed to be executed by an HPX thread after any of the registered pre-startup functions are executed (see: hpx::register_pre_startup_function()), but before *hpx_main* is being called.

This function is one of the few API functions which can be called before the runtime system has been fully initialized. It will automatically stage the provided startup function to the runtime system during its initialization (if necessary).

See also:

hpx::register_pre_startup_function()

Note: If this function is called while the startup functions are being executed or after that point, it will raise an invalid_status exception.

Parameters **f** – [in] The function to be registered to run by an HPX thread as a startup function.

hpx/runtime_local/thread_hooks.hpp

Defined in header hpx/runtime_local/thread_hooks.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

threads::*policies*::callback_notifier::on_startstop_type **get_thread_on_start_func**()

Retrieve the currently installed start handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered start function chains into the previous one (see *register_thread_on_start_func*).

Note: This function can be called before the HPX runtime is initialized.

Returns The currently installed error handler function.

threads::*policies*::callback_notifier::on_startstop_type **get_thread_on_stop_func**()

Retrieve the currently installed stop handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered stop function chains into the previous one (see *register_thread_on_stop_func*).

Note: This function can be called before the HPX runtime is initialized.

Returns The currently installed error handler function.

threads::*policies*::callback_notifier::on_error_type **get_thread_on_error_func**()

Retrieve the currently installed error handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered error function chains into the previous one (see *register_thread_on_error_func*).

Note: This function can be called before the HPX runtime is initialized.

Returns The currently installed error handler function.

threads::*policies*::callback_notifier::on_startstop_type **register_thread_on_start_func**(*threads*::*policies*::callback_notifier: &&f)

Set the currently installed start handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered start function chains into the previous one (see *get_thread_on_start_func*).

Note: This function can be called before the HPX runtime is initialized.

Parameters `f` – The function to install as the new start handler.

Returns The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

threads::*policies*::callback_notifier::on_startstop_type **`register_thread_on_stop_func`**(*threads*::*policies*::callback_notifier::o &&f)

Set the currently installed stop handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered stop function chains into the previous one (see *get_thread_on_stop_func*).

Note: This function can be called before the HPX runtime is initialized.

Parameters `f` – The function to install as the new stop handler.

Returns The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

threads::*policies*::callback_notifier::on_error_type **`register_thread_on_error_func`**(*threads*::*policies*::callback_notifier::on &&f)

Set the currently installed error handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered error function chains into the previous one (see *get_thread_on_error_func*).

Note: This function can be called before the HPX runtime is initialized.

Parameters `f` – The function to install as the new error handler.

Returns The previously registered function of this category. It is the user's responsibility to call that function if the callback is invoked by HPX.

hpx/runtime_local/thread_pool_helpers.hpp

Defined in header hpx/runtime_local/thread_pool_helpers.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **resource**

Functions

std::size_t **get_num_thread_pools**()

Return the number of thread pools currently managed by the *resource_partitioner*

std::size_t **get_num_threads**()

Return the number of threads in all thread pools currently managed by the *resource_partitioner*

std::size_t **get_num_threads**(*std*::string const &pool_name)

Return the number of threads in the given thread pool currently managed by the *resource_partitioner*

std::size_t **get_num_threads**(*std*::size_t pool_index)

Return the number of threads in the given thread pool currently managed by the *resource_partitioner*

std::size_t **get_pool_index**(*std*::string const &pool_name)

Return the internal index of the pool given its name.

std::string const &**get_pool_name**(*std*::size_t pool_index)

Return the name of the pool given its internal index.

threads::*thread_pool_base* &**get_thread_pool**(*std*::string const &pool_name)

Return the name of the pool given its name.

threads::*thread_pool_base* &**get_thread_pool**(*std*::size_t pool_index)

Return the thread pool given its internal index.

bool **pool_exists**(*std*::string const &pool_name)

Return true if the pool with the given name exists.

bool **pool_exists**(*std*::size_t pool_index)

Return true if the pool with the given index exists.

namespace **threads**

Functions

std::int64_t **get_thread_count**(*thread_schedule_state* state = *thread_schedule_state*::*unknown*)

The function *get_thread_count* returns the number of currently known threads.

Note: If state == unknown this function will not only return the number of currently existing threads, but will add the number of registered task descriptions (which have not been converted into threads yet).

Parameters state – [in] This specifies the thread-state for which the number of threads should be retrieved.

std::int64_t **get_thread_count**(*thread_priority* priority, *thread_schedule_state* state = *thread_schedule_state*::*unknown*)

The function *get_thread_count* returns the number of currently known threads.

Note: If state == unknown this function will not only return the number of currently existing threads, but will add the number of registered task descriptions (which have not been converted into threads yet).

Parameters

- **priority** – [in] This specifies the thread-priority for which the number of threads should be retrieved.
- **state** – [in] This specifies the thread-state for which the number of threads should be retrieved.

std::int64_t **get_idle_core_count**()

The function *get_idle_core_count* returns the number of currently idling threads (cores).

mask_type **get_idle_core_mask**()

The function *get_idle_core_mask* returns a bit-mask representing the currently idling threads (cores).

bool **enumerate_threads**(*hpx*::*function*<bool(thread_id_type)> const &f, *thread_schedule_state* state = *thread_schedule_state*::*unknown*)

The function *enumerate_threads* will invoke the given function *f* for each thread with a matching thread state.

Parameters

- **f** – [in] The function which should be called for each matching thread. Returning 'false' from this function will stop the enumeration process.
- **state** – [in] This specifies the thread-state for which the threads should be enumerated.

serialization

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/serialization/base_object.hpp

Defined in header hpx/serialization/base_object.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<typename **Derived**, typename **Base**>

struct *hpx*::*serialization*::**base_object_type**<*Derived*, *Base*, *std*::enable_if_t<*hpx*::*traits*::is_intrusive_polymorphic_v<*Derived*>>>

Public Functions

inline explicit constexpr **base_object_type**(*Derived* &d) noexcept

template<typename **Archive**>
inline void **save**(*Archive* &ar, unsigned) const

template<typename **Archive**>
inline void **load**(*Archive* &ar, unsigned)

HPX_SERIALIZATION_SPLIT_MEMBER()

Public Members

Derived &**d_**

namespace **hpx**

namespace **serialization**

Functions

template<typename **Base**, typename **Derived**>
constexpr *base_object_type*<*Derived*, *Base*> **base_object**(*Derived* &d) noexcept

template<typename **D**, typename **B**>
output_archive &**operator<<**(output_archive &ar, *base_object_type*<*D*, *B*> t)

template<typename **D**, typename **B**>
input_archive &**operator>>**(input_archive &ar, *base_object_type*<*D*, *B*> t)

template<typename **D**, typename **B**>
output_archive &**operator&**(output_archive &ar, *base_object_type*<*D*, *B*> t)

template<typename **D**, typename **B**>
input_archive &**operator&**(input_archive &ar, *base_object_type*<*D*, *B*> t)

template<typename **Derived**, typename **Base**, typename **Enable** = void>

struct **base_object_type**

Public Functions

inline explicit constexpr **base_object_type**(*Derived* &d) noexcept

template<typename **Archive**>
inline void **serialize**(*Archive* &ar, unsigned)

Public Members

Derived &**d_**

template<typename Derived,
typename Base> is_intrusive_polymorphic_v< Derived > > >

Public Functions

inline explicit constexpr **base_object_type**(Derived &d) noexcept

template<typename **Archive**>
inline void **save**(*Archive* &ar, unsigned) const

template<typename **Archive**>
inline void **load**(*Archive* &ar, unsigned)

HPX_SERIALIZATION_SPLIT_MEMBER()

Public Members

Derived &**d_**

synchronization

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::barrier

Defined in header hpx/barrier.hpp[722].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

template<typename **OnCompletion** = detail::empty_oncompletion>

class **barrier**

#include <barrier.hpp> A barrier is a thread coordination mechanism whose lifetime consists of a sequence of barrier phases, where each phase allows at most an expected number of threads to block until the expected number of threads arrive at the barrier. [Note: A barrier is useful for managing repeated tasks that are handled by multiple threads. - end note] Each barrier phase consists of the following steps:

- The expected count is decremented by each call to arrive or arrive_and_drop.
- When the expected count reaches zero, the phase completion step is run. For the specialization with the default value of the CompletionFunction template parameter, the completion step is run as part of the call to arrive or arrive_and_drop that caused the expected count to reach zero. For other specializations, the completion step is run on one of the threads that arrived at the barrier during the phase.
- When the completion step finishes, the expected count is reset to what was specified by the expected argument to the constructor, possibly adjusted by calls to arrive_and_drop, and the next phase starts.

Each phase defines a phase synchronization point. Threads that arrive at the barrier during the phase can block on the phase synchronization point by calling wait, and will remain blocked until the phase completion step is run. The phase completion step that is executed at the end of each phase has the following effects:

[722] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/barrier.hpp

- Invokes the completion function, equivalent to completion().
- Unblocks all threads that are blocked on the phase synchronization point.

The end of the completion step strongly happens before the returns from all calls that were unblocked by the completion step. For specializations that do not have the default value of the CompletionFunction template parameter, the behavior is undefined if any of the barrier object's member functions other than wait are called while the completion step is in progress.

Concurrent invocations of the member functions of barrier, other than its destructor, do not introduce data races. The member functions arrive and arrive_and_drop execute atomically.

CompletionFunction shall meet the Cpp17MoveConstructible (Table 28) and Cpp17Destructible (Table 32) requirements. std::is_nothrow_invocable_v<CompletionFunction&> shall be true.

The default value of the CompletionFunction template parameter is an unspecified type, such that, in addition to satisfying the requirements of CompletionFunction, it meets the Cpp17DefaultConstructible requirements (Table 27) and completion() has no effects.

barrier::arrival_token is an unspecified type, such that it meets the Cpp17MoveConstructible (Table 28), Cpp17MoveAssignable (Table 30), and Cpp17Destructible (Table 32) requirements.

Public Types

using **arrival_token** = bool

Public Functions

inline explicit constexpr **barrier**(*std*::ptrdiff_t expected, *OnCompletion* completion = *OnCompletion*())

Preconditions: expected >= 0 is true and expected <= max() is true.

Effects: Sets both the initial expected count for each barrier phase and the current expected count for the first phase to expected. Initializes completion with std::move(f). Starts the first phase. [Note: If expected is 0 this object can only be destroyed.- end note]

Throws: Any exception thrown by CompletionFunction's move constructor.

~barrier() = default

inline *arrival_token* **arrive**(*std*::ptrdiff_t update = 1)

Preconditions: update > 0 is true, and update is less than or equal to the expected count for the current barrier phase.

Effects: Constructs an object of type arrival_token that is associated with the phase synchronization point for the current phase. Then, decrements the expected count by update.

Synchronization: The call to arrive strongly happens before the start of the phase completion step for the current phase.

Error conditions: Any of the error conditions allowed for mutex types([thread.mutex.requirements.mutex]). [Note: This call can cause the completion step for the current phase to start.- end note]

Throws `system_error` – when an exception is required ([thread.req.exception]).

Returns : The constructed arrival_token object.

inline void **wait**(*arrival_token* &&old_phase) const

Preconditions: arrival is associated with the phase synchronization point for the current phase or the immediately preceding phase of the same barrier object.

Effects: Blocks at the synchronization point associated with HPX_MOVE(arrival) until the phase completion step of the synchronization point's phase is run. [Note: If arrival is associated with the synchronization point for a previous phase, the call returns immediately. - end note]

Throws `system_error` – when an exception is required ([thread.req.exception]). Error conditions: Any of the error conditions allowed for mutex types ([thread.mutex.requirements.mutex]).

inline void **arrive_and_wait**()

Effects: Equivalent to: wait(arrive()).

inline void **arrive_and_drop**()

Preconditions: The expected count for the current barrier phase is greater than zero.

Effects: Decrements the initial expected count for all subsequent phases by one. Then decrements the expected count for the current phase by one.

Synchronization: The call to arrive_and_drop strongly happens before the start of the phase completion step for the current phase.

Throws `system_error` – when an exception is required ([thread.req.exception]).Error conditions: Any of the error conditions allowed for mutex types ([thread.mutex.requirements.mutex]). [Note: This call can cause the completion step for the current phase to start.- end note]

Public Static Functions

static inline constexpr std::ptrdiff_t() max () noexcept

Private Types

using **mutex_type** = *hpx*::*spinlock*

Private Members

hpx::intrusive_ptr<detail::barrier_data> **mtx_**

mutable *hpx*::*lcos*::*local*::detail::condition_variable **cond_**

std::ptrdiff_t **expected_**

std::ptrdiff_t **arrived_**

OnCompletion **completion_**

bool **phase_**

hpx::binary_semaphore

Defined in header hpx/semaphore.hpp[723].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

class **binary_semaphore**

#include <binary_semaphore.hpp> A binary semaphore is a semaphore object that has only two states. binary_semaphore is an alias for specialization of hpx::counting_semaphore with *LeastMaxValue* being 1. HPX's implementation of binary_semaphore is more efficient than the default implementation of a counting semaphore with a unit resource count (hpx::counting_semaphore).

Public Functions

binary_semaphore(*binary_semaphore* const&) = delete

binary_semaphore &**operator=**(*binary_semaphore* const&) = delete

binary_semaphore(*binary_semaphore*&&) = delete

binary_semaphore &**operator=**(*binary_semaphore*&&) = delete

explicit **binary_semaphore**(*std*::ptrdiff_t value = 1)

Constructs an object of type hpx::binary_semaphore with the internal counter initialized to *value*.

Parameters **value** – The initial value of the internal semaphore lock count. Normally this value should be zero (which is the default), values greater than zero are equivalent to the same number of signals pre-set, and negative values are equivalent to the same number of waits pre-set.

~binary_semaphore() = default

void **release**(*std*::ptrdiff_t update = 1)

Atomically increments the internal counter by the value of update. Any thread(s) waiting for the counter to be greater than 0, such as due to being blocked in acquire, will subsequently be unblocked.

Note: Synchronization: Strongly happens before invocations of *try_acquire* that observe the result of the effects.

Throws `std::system_error` –

Parameters **update** – the amount to increment the internal counter by

Pre Both `update >= 0` and `update <= max() - counter` are *true*, where *counter* is the value of the internal counter.

bool **try_acquire**() noexcept

Tries to atomically decrement the internal counter by 1 if it is greater than 0; no blocking occurs regardless.

Returns *true* if it decremented the internal counter, otherwise *false*

[723] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/semaphore.hpp

void **acquire**()

Repeatedly performs the following steps, in order:

- Evaluates try_acquire. If the result is true, returns.

Blocks on *this until counter is greater than zero.

Throws `std::system_error` –

Returns *void*.

bool **try_acquire_until**(*hpx*::*chrono*::steady_time_point const &abs_time)

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the *abs_time* time point has been passed.

Parameters **abs_time** – the earliest time the function must wait until in order to fail

Throws `std::system_error` –

Returns *true* if it decremented the internal counter, otherwise *false*.

bool **try_acquire_for**(*hpx*::*chrono*::steady_duration const &rel_time)

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the *rel_time* duration has been exceeded.

Throws `std::system_error` –

Parameters **rel_time** – the minimum duration the function must wait for to fail

Returns *true* if it decremented the internal counter, otherwise false

Public Static Functions

static constexpr *std*::ptrdiff_t **max**() noexcept

Returns The maximum value of counter. This value is greater than or equal to *LeastMaxValue*.

Returns The internal counter's maximum possible value, as a *std::ptrdiff_t*.

hpx::condition_variable, hpx::condition_variable_any, hpx::cv_status

Defined in header hpx/condition_variable.hpp[724].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Enums

enum class **cv_status**

The scoped enumeration *hpx::cv_status* describes whether a timed wait returned because of timeout or not. *hpx::cv_status* is used by the *wait_for* and *wait_until* member functions of hpx::condition_variable and hpx::condition_variable_any.

Values:

[724] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/condition_variable.hpp

enumerator **no_timeout**

The condition variable was awakened with *notify_all*, *notify_one*, or spuriously

enumerator **timeout**

the condition variable was awakened by timeout expiration

enumerator **error**

there was an error

class **condition_variable**

#include <condition_variable.hpp> The condition_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the condition), and notifies the condition_variable.

The thread that intends to modify the shared variable has to

i. acquire a hpx::mutex (typically via *std::lock_guard*)

ii. perform the modification while the lock is held

iii. execute *notify_one* or *notify_all* on the *condition_variable* (the lock does not need to be held for notification)

Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread. Any thread that intends to wait on condition_variable has to

i. acquire a *std::unique_lock<hpx::mutex>*, on the same mutex as used to protect the shared variable

ii. either

   A. check the condition, in case it was already updated and notified

   B. execute *wait*, *await_for*, or *wait_until*. The wait operations atomically release the mutex and suspend the execution of the thread.

   C. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious. or

   A. use the predicated overload of *wait*, *wait_for*, and *wait_until*, which takes care of the three steps above.

hpx::condition_variable works only with *std::unique_lock<hpx::mutex>*. This restriction allows for maximal efficiency on some platforms. hpx::condition_variable_any provides a condition variable that works with any BasicLockable[725] object, such as *std::shared_lock*.

Condition variables permit concurrent invocation of the *wait*, *wait_for*, *wait_until*, *notify_one* and *notify_all* member functions.

The class hpx::condition_variable is a StandardLayoutType[726]. It is not CopyConstructible[727], MoveConstructible[728], CopyAssignable[729], or MoveAssignable[730].

Public Functions

inline **condition_variable**()

Construct an object of type hpx::condition_variable.

~condition_variable() = default

Destroys the object of type hpx::condition_variable.

IOW, ~condition_variable() can execute before a signaled thread returns from a wait. If this happens with condition_variable, that waiting thread will attempt to lock the destructed mutex. To fix this, there must be shared ownership of the data members between the condition_variable object and the member functions *wait* (*wait_for*, etc.).

Note: Preconditions: There is no thread blocked on *this. [Note: That is, all threads have been notified; they could subsequently block on the lock specified in the wait.This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction.The user should take care to ensure that no threads wait on *this once the destructor has been started, especially when the waiting threads are calling the *wait* functions in a loop or using the overloads of *wait*, *wait_for*, or *wait_until* that take a predicate. end note]

---

condition_variable(*condition_variable* const&) = delete

condition_variable(*condition_variable*&&) = delete

condition_variable &**operator=**(*condition_variable* const&) = delete

condition_variable &**operator=**(*condition_variable*&&) = delete

inline void **notify_one**(*error_code* &ec = *throws*) const

If any threads are waiting on **this*, calling *notify_one* unblocks one of the waiting threads.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns *notify_one* returns *void*.

inline void **notify_all**(*error_code* &ec = *throws*) const

Unblocks all threads currently waiting for **this*.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns *notify_all* returns *void*.

template<typename **Mutex**>
inline void **wait**(*std*::unique_lock<*Mutex*> &lock, *error_code* &ec = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(pred())==true).

Atomically unlocks lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when notify_all() or notify_one() is executed. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and wait exits.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters **Mutex** – Type of mutex to wait on.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns *wait* returns *void*.

template<typename **Mutex**, typename **Predicate**>
inline void **wait**(*std*::unique_lock<*Mutex*> &lock, *Predicate* pred, *error_code*& = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(pred())==true).

Equivalent to

```
while (!pred()) {
    wait(lock);
}
```

This overload may be used to ignore spurious awakenings while waiting for a specific condition to become true. Note that lock must be acquired before entering this method, and it is reacquired after *wait(lock)* exits, which means that lock can be used to guard access to *pred()*.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters

- **Mutex** – Type of mutex to wait on.
- **Predicate** – Type of predicate *pred* function.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **pred** – Predicate which returns *false* if the waiting should be continued (bool(pred())==false). The signature of the predicate function should be equivalent to the following: `bool pred();`

Returns *wait* returns *void*.

template<typename **Mutex**>
inline *cv_status* **wait_until**(*std*::unique_lock<*Mutex*> &lock, *hpx*::*chrono*::steady_time_point const &abs_time, *error_code* &ec = *throws*)

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(pred())==true).

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when notify_all() or notify_one() is executed, or when the absolute time point *abs_time* is reached. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and *wait_until* exits.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters **Mutex** – Type of mutex to wait on.

Parameters
- **lock** – *unique_lock* that must be locked by the current thread
- **abs_time** – Represents the time when waiting should be stopped
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns cv_status *wait_until* returns *hpx::cv_status::timeout* if the absolute timeout specified by *abs_time* was reached and *hpx::cv_status::no_timeout* otherwise.

template<typename **Mutex**, typename **Predicate**>
inline bool **wait_until**(*std*::unique_lock<*Mutex*> &lock, *hpx*::*chrono*::steady_time_point const &abs_time, *Predicate* pred, *error_code* &ec = *throws*)

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(pred())==true).

Equivalent to

```
while (!pred()) {
    if (wait_until(lock, abs_time) == hpx::cv_status::timeout) {
        return pred();
    }
}
return true;
```

This overload may be used to ignore spurious wakeups.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters
- **Mutex** – Type of mutex to wait on.
- **Predicate** – Type of predicate *pred* function.

Parameters
- **lock** – *unique_lock* that must be locked by the current thread
- **abs_time** – Represents the time when waiting should be stopped
- **pred** – Predicate which returns *false* if the waiting should be continued (bool(pred())==false). The signature of the predicate function should be equivalent to the following: `bool pred();`
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *wait_until* returns *false* if the predicate *pred* still evaluates to false after the *abs_time* timeout has expired, otherwise *true*. If the timeout had already expired, evaluates and returns the result of *pred*.

template<typename **Mutex**>
inline *cv_status* **wait_for**(*std*::unique_lock<*Mutex*> &lock, *hpx*::*chrono*::steady_duration const &rel_time, *error_code* &ec = *throws*)

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when notify_all() or notify_one() is executed, or when the relative timeout *rel_time* expires. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and wait_for() exits.

The standard recommends that a steady clock be used to measure the duration. This function may block for longer than *rel_time* due to scheduling or resource contention delays.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

B. Even if notified under lock, this overload makes no guarantees about the state of the associated predicate when returning due to timeout.

Template Parameters **Mutex** – Type of mutex to wait on.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **rel_time** – represents the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns cv_status *hpx::cv_status::timeout* if the relative timeout specified by *rel_time* expired, *hpx::cv_status::no_timeout* otherwise.

template<typename **Mutex**, typename **Predicate**>
inline bool **wait_for**(*std*::unique_lock<*Mutex*> &lock, *hpx*::*chrono*::steady_duration const &rel_time, *Predicate* pred, *error_code* &ec = *throws*)

Equivalent to.

```
return wait_until(lock,
          hpx::chrono::steady_clock::now() + rel_time,
          hpx::move(pred));
```

This overload may be used to ignore spurious awakenings by looping until some predicate is satisfied (bool(pred())==true).

The standard recommends that a steady clock be used to measure the duration. This function may block for longer than *rel_time* due to scheduling or resource contention delays.

Note: 1. Calling this function if *lock.mutex()* is not locked by the current thread is undefined behavior.

A. Calling this function if *lock.mutex()* is not the same mutex as the one used by all other threads that are currently waiting on the same condition variable is undefined behavior.

Template Parameters

- **Mutex** – Type of mutex to wait on.
- **Predicate** – Type of predicate *pred* function.

Parameters

- **lock** – *unique_lock* that must be locked by the current thread
- **rel_time** – represents the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **pred** – Predicate which returns *false* if the waiting should be continued (bool(pred())==false). The signature of the predicate function should be equivalent to the following: `bool pred();`
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *wait_for* returns *false* if the predicate pred still evaluates to *false* after the *rel_time* timeout expired, otherwise *true*.

Private Types

using **mutex_type** = *lcos*::*local*::detail::condition_variable_data::mutex_type

using **data_type** = *hpx*::intrusive_ptr<*lcos*::*local*::detail::condition_variable_data>

Private Members

hpx::*util*::cache_aligned_data_derived<*data_type*> **data_**

class **condition_variable_any**

#include <condition_variable.hpp> The condition_variable_any class is a generalization of hpx::condition_variable. Whereas hpx::condition_variable works only on *std::unique_lock<std::mutex>*, *a condition_variable_any* can operate on any lock that meets the BasicLockable[731] requirements.

See hpx::condition_variable for the description of the semantics of condition variables. It is not CopyConstructible[732], MoveConstructible[733], CopyAssignable[734], or MoveAssignable[735].

Public Functions

inline **condition_variable_any**()

Constructs an object of type hpx::condition_variable_any.

~condition_variable_any() = default

Destroys the object of type hpx::condition_variable_any.

It is only safe to invoke the destructor if all threads have been notified. It is not required that they have exited their respective wait functions: some threads may still be waiting to reacquire the associated lock, or may be waiting to be scheduled to run after reacquiring it.

The programmer must ensure that no threads attempt to wait on **this* once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or are using the overloads of the wait functions that take a predicate.

Preconditions: There is no thread blocked on *this. [Note: That is, all threads have been notified; they could subsequently block on the lock specified in the wait.This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. The user should take care to ensure that no threads wait on **this* once the destructor has been started, especially when the waiting threads are calling the *wait* functions in a loop or using the overloads of *wait*, *wait_for*, or *wait_until* that take a predicate. end note]

IOW, ~condition_variable_any() can execute before a signaled thread returns from a wait. If this happens with condition_variable_any, that waiting thread will attempt to lock the destructed mutex. To fix this, there must be shared ownership of the data members between the condition_variable_any object and the member functions wait (*wait_for*, etc.).

condition_variable_any(*condition_variable_any* const&) = delete

condition_variable_any(*condition_variable_any*&&) = delete

condition_variable_any &**operator=**(*condition_variable_any* const&) = delete

condition_variable_any &**operator=**(*condition_variable_any*&&) = delete

inline void **notify_one**(*error_code* &ec = *throws*) const

If any threads are waiting on **this*, calling *notify_one* unblocks one of the waiting threads.

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock. However, some implementations (in particular many implementations of pthreads) recognize this situation and avoid this "hurry up and wait" scenario by transferring the waiting thread from the condition variable's queue directly to the queue of the mutex within the notify call, without waking it up.

Notifying while under the lock may nevertheless be necessary when precise scheduling of events is required, e.g. if the waiting thread would exit the program if the condition is satisfied, causing destruction of the notifying thread's condition variable. A spurious wakeup after mutex unlock but before notify would result in notify called on a destroyed object.

Note: The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns *notify_one* returns *void*.

inline void **notify_all**(*error_code* &ec = *throws*) const

Unblocks all threads currently waiting for **this*.

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock.

Note: The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns *notify_all* returns *void*.

template<typename **Lock**>
inline void **wait**(*Lock* &lock, *error_code* &ec = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(pred())==true).

Atomically unlocks lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when notify_all() or notify_one() is executed. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and wait exits.

The effects of *notify_one()*/notify_all() and each of the three atomic parts of *wait()*/wait_for()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Note: If these functions fail to meet the postconditions (lock is locked by the calling thread), std::terminate is called. For example, this could happen if re-locking the mutex throws an exception.

Template Parameters **Lock** – Type of *lock*.

Parameters
- **lock** – An object of type Lock that meets the BasicLockable[736] requirements, which must be locked by the current thread
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special'throw on error' *error_code*.

Returns *wait* returns *void*.

template<typename **Lock**, typename **Predicate**>
inline void **wait**(*Lock* &lock, *Predicate* pred, *error_code*& = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(pred())==true).

Equivalent to

```
while (!pred()) {
    wait(lock);
}
```

This overload may be used to ignore spurious awakenings while waiting for a specific condition to become true. Note that lock must be acquired before entering this method, and it is reacquired after *wait(lock)* exits, which means that lock can be used to guard access to *pred()*.

The effects of *notify_one()*/notify_all() and each of the three atomic parts of *wait()*/wait_for()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Note: If these functions fail to meet the postconditions (lock is locked by the calling thread), std::terminate is called. For example, this could happen if re-locking the mutex throws an exception.

Template Parameters
- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters
- **lock** – an object of type Lock that meets the BasicLockable[737] requirements, which must be locked by the current thread

- **pred** – predicate which returns `false` if the waiting should be continued (`bool(pred()) == false`). The signature of the predicate function should be equivalent to the following: `bool pred()`.

Returns *wait* returns *void*.

template<typename **Lock**>
inline *cv_status* **wait_until**(*Lock* &lock, *hpx*::*chrono*::steady_time_point const &abs_time, *error_code* &ec = *throws*)

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (`bool(pred()) == true`).

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when notify_all() or notify_one() is executed, or when the absolute time point *abs_time* is reached. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and *wait_until* exits.

Note: The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Template Parameters **Lock** – Type of *lock*.

Parameters

- **lock** – an object of type *Lock* that meets the requirements of BasicLockable[738], which must be locked by the current thread
- **abs_time** – represents the time when waiting should be stopped.
- **ec** – used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns cv_status *hpx::cv_status::timeout* if the absolute timeout specified by *abs_time* was reached, *hpx::cv_status::no_timeout* otherwise.

template<typename **Lock**, typename **Predicate**>
inline bool **wait_until**(*Lock* &lock, *hpx*::*chrono*::steady_time_point const &abs_time, *Predicate* pred, *error_code* &ec = *throws*)

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied (`bool(pred()) == true`).

Equivalent to

```
while (!pred()) {
    if (wait_until(lock, timeout_time) == hpx::cv_status::timeout) {
       return pred();
    }
}
return true;
```

This overload may be used to ignore spurious wakeups.

Note: The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Template Parameters
- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters
- **lock** – an object of type *Lock* that meets the requirements of BasicLockable[739], which must be locked by the current thread
- **abs_time** – represents the time when waiting should be stopped.
- **pred** – predicate which returns *false* if the waiting should be continued `(bool(pred()) == false)`. The signature of the predicate function should be equivalent to the following: `bool pred();`.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *false* if the predicate *pred* still evaluates to *false* after the *abs_time* timeout expired, otherwise true. If the timeout had already expired, evaluates and returns the result of *pred*.

template<typename **Lock**>
inline *cv_status* **wait_for**(*Lock* &lock, *hpx*::*chrono*::steady_duration const &rel_time, *error_code* &ec = *throws*)

Atomically releases lock, blocks the current executing thread, and adds it to the list of threads waiting on **this*. The thread will be unblocked when notify_all() or notify_one() is executed, or when the relative timeout *rel_time* expires. It may also be unblocked spuriously. When unblocked, regardless of the reason, *lock* is reacquired and wait_for() exits.

The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Note: Even if notified under lock, this overload makes no guarantees about the state of the associated predicate when returning due to timeout.

Template Parameters **Lock** – Type of *lock*.

Parameters
- **lock** – an object of type *Lock* that meets the BasicLockable[740] requirements, which must be locked by the current thread.
- **rel_time** – an object of type *hpx::chrono::duration* representing the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns cv_status *hpx::cv_status::timeout* if the relative timeout specified by *rel_time* expired, *hpx::cv_status::no_timeout* otherwise.

template<typename **Lock**, typename **Predicate**>
inline bool **wait_for**(*Lock* &lock, *hpx*::*chrono*::steady_duration const &rel_time, *Predicate* pred, *error_code* &ec = *throws*)

Equivalent to.

```
return wait_until(lock,
    hpx::chrono::steady_clock::now() + rel_time,
    std::move(pred));
```

This overload may be used to ignore spurious awakenings by looping until some predicate is satisfied (`bool(pred()) == true`).

Note: The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Template Parameters
- **`Lock`** – Type of *lock*.
- **`Predicate`** – Type of *pred*.

Parameters
- **`lock`** – an object of type *Lock* that meets the BasicLockable[741] requirements,which must be locked by the current thread.
- **`rel_time`** – an object of type *hpx::chrono::duration* representing the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **`pred`** – predicate which returns *false* if the waiting should be continued (`bool(pred()) == false`). The signature of the predicate function should be equivalent to the following: `bool pred();`.
- **`ec`** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *false* if the predicate *pred* still evaluates to *false* after the *rel_time* timeout expired, otherwise *true*.

template<typename **Lock**, typename **Predicate**>
inline bool **wait**(*Lock* &lock, *stop_token* stoken, *Predicate* pred, *error_code* &ec = *throws*)

wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied (bool(pred())==true).

An interruptible wait: registers the condition_variable_any for the duration of wait(), to be notified if a stop request is made on the given stoken's associated stop-state; it is then equivalent to

```
while (!stoken.stop_requested()) {
    if (pred()) return true;
    wait(lock);
}
return pred();
```

Note that the returned value indicates whether *pred* evaluated to *true*, regardless of whether there was a stop requested or not.

Note: The effects of `notify_one()`/`notify_all()` and each of the three atomic parts of `wait()`/`wait_for()`/`wait_until()` (`unlock+wait, wakeup, and lock`) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for *notify_one()* to, for example, be delayed and unblock a thread that started waiting just after the call to `notify_one()` was made.

Template Parameters

- **`Lock`** – Type of *lock*.
- **`Predicate`** – Type of *pred*.

Parameters

- **`lock`** – an object of type Lock that meets the BasicLockable[742] requirements, which must be locked by the current thread
- **`stoken`** – a hpx::stop_token to register interruption for
- **`pred`** – predicate which returns `false` if the waiting should be continued `(bool(pred()) == false)`. The signature of the predicate function should be equivalent to the following: `bool pred()`.
- **`ec`** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool result of *pred()*.

template<typename **`Lock`**, typename **`Predicate`**>
inline bool **`wait_until`**(*Lock* &lock, *stop_token* stoken, *hpx*::*chrono*::steady_time_point const &abs_time, *Predicate* pred, *error_code* &ec = *throws*)

wait_until causes the current thread to block until the condition variable is notified, a specific time is reached, or a spurious wakeup occurs, optionally looping until some predicate is satisfied `(bool(pred()) == true)`.

An interruptible wait: registers the condition_variable_any for the duration of wait_until(), to be notified if a stop request is made on the given stoken's associated stop-state; it is then equivalent to

```
while (!stoken.stop_requested()) {
    if (pred())
        return true;
    if (wait_until(lock, timeout_time) == hpx::cv_status::timeout)
        return pred();
}
return pred();
```

Note: The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Template Parameters

- **`Lock`** – Type of *lock*.
- **`Predicate`** – Type of *pred*.

Parameters

- **`lock`** – an object of type *Lock* that meets the requirements of BasicLockable[743], which must be locked by the current thread.
- **`stoken`** – a hpx::stop_token to register interruption for.
- **`abs_time`** – represents the time when waiting should be stopped.
- **`pred`** – predicate which returns *false* if the waiting should be continued `(bool(pred()) == false)`. The signature of the predicate function should be equivalent to the following: `bool pred();`.

- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *pred()*, regardless of whether the timeout was met or stop was requested.

template<typename **Lock**, typename **Predicate**>
inline bool **wait_for**(*Lock* &lock, *stop_token* stoken, *hpx*::*chrono*::steady_duration const &rel_time, *Predicate* pred, *error_code* &ec = *throws*)

Equivalent to.

```
return wait_until(lock, std::move(stoken),
    hpx::chrono::steady_clock::now() + rel_time,
    std::move(pred));
```

Note: The effects of notify_one()/*notify_all*() and each of the three atomic parts of wait()/*wait_for*()/wait_until() (unlock+wait, wakeup, and lock) take place in a single total order that can be viewed as modification order of an atomic variable: the order is specific to this individual condition variable. This makes it impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

Template Parameters
- **Lock** – Type of *lock*.
- **Predicate** – Type of *pred*.

Parameters
- **lock** – an object of type *Lock* that meets the BasicLockable[744] requirements, which must be locked by the current thread.
- **stoken** – a hpx::stop_token to register interruption for.
- **rel_time** – an object of type *hpx::chrono::duration* representing the maximum time to spend waiting. Note that *rel_time* must be small enough not to overflow when added to *hpx::chrono::steady_clock::now()*.
- **pred** – predicate which returns *false* if the waiting should be continued `(bool(pred()) == false)`. The signature of the predicate function should be equivalent to the following: `bool pred();`.
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *pred()*, regardless of whether the timeout was met or stop was requested.

Private Types

using **mutex_type** = *lcos*::*local*::detail::condition_variable_data::mutex_type

using **data_type** = *hpx*::intrusive_ptr<*lcos*::*local*::detail::condition_variable_data>

Private Members

hpx::*util*::cache_aligned_data_derived<*data_type*> **data_**

namespace **lcos**

namespace **local**

Typedefs

typedef *hpx*::*condition_variable* **instead**

hpx::counting_semaphore

Defined in header hpx/semaphore.hpp[745].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

template<*std*::ptrdiff_t **LeastMaxValue** = PTRDIFF_MAX>

class **counting_semaphore**

#include <counting_semaphore.hpp> A semaphore is a protected variable (an entity storing a value) or abstract data type (an entity grouping several variables that may or may not be numerical) which constitutes the classic method for restricting access to shared resources, such as shared memory, in a multiprogramming environment. Semaphores exist in many variants, though usually the term refers to a counting semaphore, since a binary semaphore is better known as a mutex. A counting semaphore is a counter for a set of available resources, rather than a locked/unlocked flag of a single resource. It was invented by Edsger Dijkstra. Semaphores are the classic solution to preventing race conditions in the dining philosophers problem, although they do not prevent resource deadlocks.

[725] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[726] https://en.cppreference.com/w/cpp/named_req/StandardLayoutType
[727] https://en.cppreference.com/w/cpp/named_req/CopyConstructible
[728] https://en.cppreference.com/w/cpp/named_req/MoveConstructible
[729] https://en.cppreference.com/w/cpp/named_req/CopyAssignable
[730] https://en.cppreference.com/w/cpp/named_req/MoveAssignable
[731] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[732] https://en.cppreference.com/w/cpp/named_req/CopyConstructible
[733] https://en.cppreference.com/w/cpp/named_req/MoveConstructible
[734] https://en.cppreference.com/w/cpp/named_req/CopyAssignable
[735] https://en.cppreference.com/w/cpp/named_req/MoveAssignable
[736] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[737] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[738] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[739] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[740] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[741] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[742] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[743] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[744] https://en.cppreference.com/w/cpp/named_req/BasicLockable
[745] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/semaphore.hpp

Counting semaphores can be used for synchronizing multiple threads as well: one thread waiting for several other threads to touch (signal) the semaphore, or several threads waiting for one other thread to touch this semaphore. Unlike hpx::mutex a counting_semaphore is not tied to threads of execution — acquiring a semaphore can occur on a different thread than releasing the semaphore, for example. All operations on counting_semaphore can be performed concurrently and without any relation to specific threads of execution, with the exception of the destructor which cannot be performed concurrently but can be performed on a different thread.

Semaphores are lightweight synchronization primitives used to constrain concurrent access to a shared resource. They are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.

A counting semaphore is a semaphore object that models a non-negative resource count.

Class template counting_semaphore maintains an internal counter that is initialized when the semaphore is created. The counter is decremented when a thread acquires the semaphore, and is incremented when a thread releases the semaphore. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

Specializations of hpx::counting_semaphore are not DefaultConstructible[746], CopyConstructible[747], MoveConstructible[748], CopyAssignable[749], or MoveAssignable[750].

Note: counting_semaphore*'s* try_acquire() can spuriously fail.

Template Parameters **`LeastMaxValue`** – counting_semaphore allows more than one concurrent access to the same resource, for at least *LeastMaxValue* concurrent accessors. As its name indicates, the *LeastMaxValue* is the minimum max value, not the actual max value. Thus max() can yield a number larger than *LeastMaxValue*.

Public Functions

`counting_semaphore`(*counting_semaphore* const&) = delete

counting_semaphore &**`operator=`**(*counting_semaphore* const&) = delete

`counting_semaphore`(*counting_semaphore*&&) = delete

counting_semaphore &**`operator=`**(*counting_semaphore*&&) = delete

explicit **`counting_semaphore`**(*std*::ptrdiff_t value)

Constructs an object of type hpx::counting_semaphore with the internal counter initialized to *value*.

Parameters **`value`** – The initial value of the internal semaphore lock count. Normally this value should be zero (which is the default), values greater than zero are equivalent to the same number of signals pre-set, and negative values are equivalent to the same number of waits pre-set.

`~counting_semaphore`() = default

void **`release`**(*std*::ptrdiff_t update = 1)

Atomically increments the internal counter by the value of update. Any thread(s) waiting for the counter to be greater than 0, such as due to being blocked in acquire, will subsequently be unblocked.

Note: Synchronization: Strongly happens before invocations of *try_acquire* that observe the result of the effects.

Throws *std*::system_error –

Parameters **update** – the amount to increment the internal counter by

Pre Both update >= 0 and update <= *max()* - counter are *true*, where *counter* is the value of the internal counter.

bool **try_acquire**() noexcept

Tries to atomically decrement the internal counter by 1 if it is greater than 0; no blocking occurs regardless.

Returns *true* if it decremented the internal counter, otherwise *false*

void **acquire**()

Repeatedly performs the following steps, in order:

- Evaluates try_acquire. If the result is true, returns.
- Blocks on *this until counter is greater than zero.

Throws *std*::system_error –

Returns *void*.

bool **try_acquire_until**(*hpx*::*chrono*::steady_time_point const &abs_time)

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the *abs_time* time point has been passed.

Parameters **abs_time** – the earliest time the function must wait until in order to fail

Throws *std*::system_error –

Returns *true* if it decremented the internal counter, otherwise *false*.

bool **try_acquire_for**(*hpx*::*chrono*::steady_duration const &rel_time)

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the *rel_time* duration has been exceeded.

Throws *std*::system_error –

Parameters **rel_time** – the minimum duration the function must wait for to fail

Returns *true* if it decremented the internal counter, otherwise false

Public Static Functions

static constexpr *std*::ptrdiff_t **max**() noexcept

Returns The maximum value of counter. This value is greater than or equal to *LeastMaxValue*.

Returns The internal counter's maximum possible value, as a *std::ptrdiff_t*.

template<typename **Mutex** = *hpx*::*spinlock*, int **N** = 0>

class **counting_semaphore_var**

#include <counting_semaphore.hpp> A semaphore is a protected variable (an entity storing a value) or abstract data type (an entity grouping several variables that may or may not be numerical) which constitutes the classic method for restricting access to shared resources, such as shared memory, in a multiprogramming environment. Semaphores exist in many variants, though usually the term refers to a counting semaphore, since a binary semaphore is better known as a mutex. A counting semaphore is a counter for a set of available resources, rather than a locked/unlocked flag of a single resource. It was invented by Edsger Dijkstra. Semaphores are the classic solution to preventing race conditions in the dining philosophers problem, although they do not prevent resource deadlocks.

Counting semaphores can be used for synchronizing multiple threads as well: one thread waiting for several other threads to touch (signal) the semaphore, or several threads waiting for one other thread to touch this semaphore. Unlike hpx::mutex a counting_semaphore_var is not tied to threads of execution — acquiring a semaphore can occur on a different thread than releasing the semaphore, for example. All operations on counting_semaphore_var can be performed concurrently and without any relation to specific threads of execution, with the exception of the destructor which cannot be performed concurrently but can be performed on a different thread.

Semaphores are lightweight synchronization primitives used to constrain concurrent access to a shared resource. They are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables.

A counting semaphore is a semaphore object that models a non-negative resource count.

Class template counting_semaphore_var maintains an internal counter that is initialized when the semaphore is created. The counter is decremented when a thread acquires the semaphore, and is incremented when a thread releases the semaphore. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

Specializations of hpx::counting_semaphore_var are not DefaultConstructible[751], CopyConstructible[752], MoveConstructible[753], CopyAssignable[754], or MoveAssignable[755].

Note: counting_semaphore_var*'s* try_acquire() can spuriously fail.

Template Parameters

- **`Mutex`** – Type of mutex
- **`N`** – The initial value of the internal semaphore lock count.

Public Functions

explicit **`counting_semaphore_var`**(*std*::ptrdiff_t value = *N*)

Constructs an object of type *hpx::counting_semaphore_value* with the internal counter initialized to *N*.

Parameters **`value`** – The initial value of the internal semaphore lock count. Normally this value should be zero, values greater than zero are equivalent to the same number of signals pre-set, and negative values are equivalent to the same number of waits pre-set. Defaults to *N* (which in turn defaults to zero).

`counting_semaphore_var`(*counting_semaphore_var* const&) = delete

counting_semaphore_var &**`operator=`**(*counting_semaphore_var* const&) = delete

void **`wait`**(*std*::ptrdiff_t count = 1)

Wait for the semaphore to be signaled.

Parameters **`count`** – The value by which the internal lock count will be decremented. At the same time this is the minimum value of the lock count at which the thread is not yielded.

bool **`try_wait`**(*std*::ptrdiff_t count = 1)

Try to wait for the semaphore to be signaled.

Parameters **`count`** – The value by which the internal lock count will be decremented. At the same time this is the minimum value of the lock count at which the thread is not yielded.

Returns *try_wait* returns true if the calling thread was able to acquire the requested amount of credits. *try_wait* returns false if not sufficient credits are available at this point in time.

void **signal**(*std*::ptrdiff_t count = 1)

Signal the semaphore.

Parameters **count** – The value by which the internal lock count will be incremented.

std::ptrdiff_t **signal_all**()

Unblock all acquirers.

Returns std::ptrdiff_t internal lock count after the operation.

void **release**(*std*::ptrdiff_t update = 1)

Atomically increments the internal counter by the value of update. Any thread(s) waiting for the counter to be greater than 0, such as due to being blocked in acquire, will subsequently be unblocked.

Note: Synchronization: Strongly happens before invocations of *try_acquire* that observe the result of the effects.

Throws `std::system_error` –

Parameters **update** – the amount to increment the internal counter by

Pre Both `update >= 0` and `update <= max() - counter` are *true*, where *counter* is the value of the internal counter.

bool **try_acquire**() noexcept

Tries to atomically decrement the internal counter by 1 if it is greater than 0; no blocking occurs regardless.

Returns *true* if it decremented the internal counter, otherwise *false*

void **acquire**()

Repeatedly performs the following steps, in order:

- Evaluates try_acquire. If the result is true, returns.

Blocks on *this until counter is greater than zero.

Throws `std::system_error` –

Returns *void*.

bool **try_acquire_until**(*hpx*::*chrono*::steady_time_point const &abs_time)

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the *abs_time* time point has been passed.

Parameters **abs_time** – the earliest time the function must wait until in order to fail

Throws `std::system_error` –

Returns *true* if it decremented the internal counter, otherwise *false*.

bool **try_acquire_for**(*hpx*::*chrono*::steady_duration const &rel_time)

Tries to atomically decrement the internal counter by 1 if it is greater than 0; otherwise blocks until it is greater than 0 and can successfully decrement the internal counter, or the *rel_time* duration has been exceeded.

Throws `std::system_error` –

Parameters **rel_time** – the minimum duration the function must wait for to fail

Returns *true* if it decremented the internal counter, otherwise false

Public Static Functions

static constexpr *std*::ptrdiff_t **max()** noexcept

Returns The maximum value of counter. This value is greater than or equal to *LeastMaxValue*.

Returns The internal counter's maximum possible value, as a *std::ptrdiff_t*.

Private Types

using **mutex_type** = *Mutex*

hpx/synchronization/event.hpp

Defined in header hpx/synchronization/event.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **lcos**

namespace **local**

class **event**

#include <event.hpp> Event semaphores can be used for synchronizing multiple threads that need to wait for an event to occur. When the event occurs, all threads waiting for the event are woken up.

Public Functions

inline **event()** noexcept

Construct a new event semaphore.

inline bool **occurred()** const noexcept

Check if the event has occurred.

inline void **wait()**

Wait for the event to occur.

inline void **set()**

Release all threads waiting on this semaphore.

[746] https://en.cppreference.com/w/cpp/named_req/DefaultConstructible
[747] https://en.cppreference.com/w/cpp/named_req/CopyConstructible
[748] https://en.cppreference.com/w/cpp/named_req/MoveConstructible
[749] https://en.cppreference.com/w/cpp/named_req/CopyAssignable
[750] https://en.cppreference.com/w/cpp/named_req/MoveAssignable
[751] https://en.cppreference.com/w/cpp/named_req/DefaultConstructible
[752] https://en.cppreference.com/w/cpp/named_req/CopyConstructible
[753] https://en.cppreference.com/w/cpp/named_req/MoveConstructible
[754] https://en.cppreference.com/w/cpp/named_req/CopyAssignable
[755] https://en.cppreference.com/w/cpp/named_req/MoveAssignable

inline void **reset**() noexcept

Reset the event.

Private Types

using **mutex_type** = *hpx*::*spinlock*

Private Functions

inline void **wait_locked**(*std*::unique_lock<*mutex_type*> &l)

inline void **set_locked**(*std*::unique_lock<*mutex_type*> l)

Private Members

mutex_type **mtx_**

This mutex protects the queue.

local::detail::condition_variable **cond_**

std::atomic<bool> **event_**

hpx::latch

Defined in header hpx/latch.hpp[756].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

class **latch**

#include <latch.hpp> Latches are a thread coordination mechanism that allow one or more threads to block until an operation is completed. An individual latch is a single-use object; once the operation has been completed, the latch cannot be reused.

Subclassed by *hpx::lcos::local::latch*

[756] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/latch.hpp

Public Functions

latch(*latch* const&) = delete

latch(*latch*&&) = delete

latch &**operator=**(*latch* const&) = delete

latch &**operator=**(*latch*&&) = delete

inline explicit **latch**(*std*::ptrdiff_t count)

Initialize the latch

Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

~latch() = default

Requires: No threads are blocked at the synchronization point.

Note: May be called even if some threads have not yet returned from *wait()* or count_down_and_wait(), provided that counter_ is 0.

Note: The destructor might not return until all threads have exited *wait()* or count_down_and_wait().

Note: It is the caller's responsibility to ensure that no other thread enters *wait()* after one thread has called the destructor. This may require additional coordination.

---

inline void **count_down**(*std*::ptrdiff_t update)

Decrements counter_ by n. Does not block.

Requires: counter_ >= n and n >= 0.

Synchronization: Synchronizes with all calls that block on this latch and with all try_wait calls on this latch that return true .

Throws `Nothing.` –

inline bool **try_wait**() const noexcept

Returns: With very low probability false. Otherwise counter == 0.

inline void **wait**() const

If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization point until counter_ reaches 0.

Throws `Nothing.` –

inline void **arrive_and_wait**(*std*::ptrdiff_t update = 1)

Effects: Equivalent to: count_down(update); *wait()*;

Public Static Functions

static inline constexpr std::ptrdiff_t() max () noexcept

Returns: The maximum value of counter that the implementation supports.

Protected Types

using **mutex_type** = *hpx*::*spinlock*

Protected Attributes

mutable *util*::cache_line_data<*mutex_type*> **mtx_**

mutable *util*::cache_line_data<*hpx*::*lcos*::*local*::detail::condition_variable> **cond_**

std::atomic<*std*::ptrdiff_t> **counter_**

bool **notified_**

namespace **lcos**

namespace **local**

class **latch** : public *hpx*::*latch*

#include <latch.hpp> A latch maintains an internal counter_ that is initialized when the latch is created. Threads may block at a synchronization point waiting for counter_ to be decremented to 0. When counter_ reaches 0, all such blocked threads are released.

Calls to countdown_and_wait() , *count_down()* , *wait()* , *is_ready()*, *count_up()* , and *reset()* behave as atomic operations.

Note: A hpx::latch is not an LCO in the sense that it has no global id and it can't be triggered using the action (parcel) mechanism. Use hpx::distributed::latch instead if this is required. It is just a low level synchronization primitive allowing to synchronize a given number of *threads*.

Public Functions

HPX_NON_COPYABLE(*latch*)

inline explicit **latch**(*std*::ptrdiff_t count)

Initialize the latch

Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

~latch() = default

Requires: No threads are blocked at the synchronization point.

Note: May be called even if some threads have not yet returned from *wait()* or *count_down_and_wait()*, provided that counter_ is 0.

Note: The destructor might not return until all threads have exited *wait()* or *count_down_and_wait()*.

Note: It is the caller's responsibility to ensure that no other thread enters *wait()* after one thread has called the destructor. This may require additional coordination.

---

inline void **count_down_and_wait**()

Decrements counter_ by 1 . Blocks at the synchronization point until counter_ reaches 0.

Requires: counter_ > 0.

Synchronization: Synchronizes with all calls that block on this latch and with all is_ready calls on this latch that return true.

Throws `Nothing.` –

inline bool **is_ready**() const noexcept

Returns: counter_ == 0. Does not block.

Throws `Nothing.` –

inline void **abort_all**() const

inline void **count_up**(*std*::ptrdiff_t n)

Increments counter_ by n. Does not block.

Requires: n >= 0.

Throws `Nothing.` –

inline void **reset**(*std*::ptrdiff_t n)

Reset counter_ to n. Does not block.

Requires: n >= 0.

Throws `Nothing.` –

inline bool **reset_if_needed_and_count_up**(*std*::ptrdiff_t n, *std*::ptrdiff_t count)

Effects: Equivalent to: if (*is_ready()*) reset(count); count_up(n); Returns: true if the latch was reset

hpx::mutex, hpx::timed_mutex

Defined in header hpx/mutex.hpp[757].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

class **mutex**

#include <mutex.hpp> mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. *mutex* offers exclusive, non-recursive ownership semantics:

- A calling thread owns a mutex from the time that it successfully calls either *lock* or *try_lock* until it calls *unlock*.
- When a thread owns a *mutex*, all other threads will block (for calls to *lock*) or receive a *false* return value (for *try_lock*) if they attempt to claim ownership of the *mutex*.
- A calling thread must not own the *mutex* prior to calling *lock* or *try_lock*.

The behavior of a program is undefined if a *mutex* is destroyed while still owned by any threads, or a thread terminates while owning a *mutex*. The mutex class satisfies all requirements of Mutex[758] and StandardLayoutType[759].

hpx::mutex is neither copyable nor movable.

Subclassed by *hpx::timed_mutex*

Public Functions

HPX_NON_COPYABLE(*mutex*)

hpx::mutex is neither copyable nor movable

inline HPX_HOST_DEVICE_CONSTEXPR **mutex**(char const*const = "") noexcept

Constructs the *mutex*. The *mutex* is in unlocked state after the constructor completes.

Note: Because the default constructor is *constexpr*, static mutexes are initialized as part of static non-local initialization, before any dynamic non-local initialization begins. This makes it safe to lock a *mutex* in a constructor of any static object.

Parameters description – description of the *mutex*.

~mutex()

Destroys the *mutex*. The behavior is undefined if the *mutex* is owned by any thread or if any thread terminates while holding any ownership of the *mutex*.

void **lock**(char const *description, *error_code* &ec = *throws*)

Locks the mutex. If another thread has already locked the mutex, a call to lock will block execution until the lock is acquired. If lock is called by a thread that already owns the mutex, the behavior is undefined: for example, the program may deadlock. hpx::mutex can detect the invalid usage and

[757] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/mutex.hpp

throws a *std::system_error* with error condition *resource_deadlock_would_occur* instead of deadlocking. Prior unlock() operations on the same mutex synchronize- with (as defined in *std::memory_order*) this operation.

Note: lock() is usually not called directly: *std::unique_lock*, *std::scoped_lock*, and *std::lock_guard* are used to manage exclusive locking.

Parameters
- **`description`** – Description of the *mutex*
- **`ec`** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns void *lock* returns *void*.

inline void **`lock`**(*error_code* &ec = *throws*)

Locks the mutex. If another thread has already locked the mutex, a call to lock will block execution until the lock is acquired. If lock is called by a thread that already owns the mutex, the behavior is undefined: for example, the program may deadlock. hpx::mutex can detect the invalid usage and throws a *std::system_error* with error condition *resource_deadlock_would_occur* instead of deadlocking. Prior unlock() operations on the same mutex synchronize - with(as defined in *std::memory_order*) this operation.

Note: lock() is usually not called directly: *std::unique_lock*, *std::scoped_lock*, and *std::lock_guard* are used to manage exclusive locking. This overload essentially calls `void lock(char const* description, error_code& ec =throws);` with *description* as `mutex::lock`.

Parameters **`ec`** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns void *lock* returns *void*.

bool **`try_lock`**(char const *description, *error_code* &ec = *throws*)

Tries to lock the *mutex*. Returns immediately. On successful lock acquisition returns *true*, otherwise returns *false*. This function is allowed to fail spuriously and return *false* even if the *mutex* is not currently locked by any other thread. If *try_lock* is called by a thread that already owns the *mutex*, the behavior is undefined. Prior unlock() operation on the same mutex synchronizes-with (as defined in *std::memory_order*) this operation if it returns *true*. Note that prior lock() does not synchronize with this operation if it returns *false*.

Parameters
- **`description`** – Description of the *mutex*
- **`ec`** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock* returns *true* on successful lock acquisition, otherwise returns *false*.

inline bool **`try_lock`**(*error_code* &ec = *throws*)

Tries to lock the *mutex*. Returns immediately. On successful lock acquisition returns *true*, otherwise returns *false*. This function is allowed to fail spuriously and return *false* even if the *mutex* is not currently locked by any other thread. If *try_lock* is called by a thread that already owns the *mutex*, the behavior is undefined. Prior unlock() operation on the same mutex synchronizes-with (as defined in *std::memory_order*) this operation if it returns *true*. Note that prior lock() does not synchronize with this operation if it returns *false*.

Note: This overload essentially calls

```
void try_lock(char const* description,
              error_code& ec = throws);
```

with *description* as `mutex::try_lock`.

Parameters `ec` – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock* returns *true* on successful lock acquisition, otherwise returns *false*.

void **unlock**(*error_code* &ec = *throws*)

Unlocks the *mutex*. The *mutex* must be locked by the current thread of execution, otherwise, the behavior is undefined. This operation *synchronizes-with* (as defined in *std::memory_order*) any subsequent *lock* operation that obtains ownership of the same *mutex*.

Parameters `ec` – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns *unlock* returns *void*.

class **timed_mutex** : private *hpx*::*mutex*

#include <mutex.hpp> The timed_mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In a manner similar to *mutex*, timed_mutex offers exclusive, non-recursive ownership semantics. In addition, timed_mutex provides the ability to attempt to claim ownership of a timed_mutex with a timeout via the member functions try_lock_for() and try_lock_until(). The timed_mutex class satisfies all requirements of TimedMutex[760] and StandardLayoutType[761].

hpx::timed_mutex is neither copyable nor movable.

Public Functions

HPX_NON_COPYABLE(*timed_mutex*)

hpx::timed_mutex is neither copyable nor movable

timed_mutex(char const *const description = "")

Constructs a timed_mutex. The mutex is in unlocked state after the call.

Parameters `description` – Description of the timed_mutex.

~timed_mutex()

Destroys the timed_mutex. The behavior is undefined if the mutex is owned by any thread or if any thread terminates while holding any ownership of the mutex.

bool **try_lock_until**(*hpx*::*chrono*::steady_time_point const &abs_time, char const *description, *error_code* &ec = *throws*)

Tries to lock the mutex. Blocks until specified *abs_time* has been reached or the lock is acquired, whichever comes first. On successful lock acquisition returns *true*, otherwise returns *false*. If *abs_time* has already passed, this function behaves like try_lock(). As with try_lock(), this function is allowed to fail spuriously and return *false* even if the mutex was not locked by any other thread at some point before *abs_time*. Prior unlock() operation on the same mutex *synchronizes-with* (as defined in *std::memory_order*) this operation if it returns *true*. If *try_lock_until* is called by a thread that already owns the mutex, the behavior is undefined.

Parameters

- `abs_time` – time point to block until
- `description` – Description of the timed_mutex

- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock_until* returns *true* if the lock was acquired successfully, otherwise *false*.

inline bool **try_lock_until**(*hpx*::*chrono*::steady_time_point const &abs_time, *error_code* &ec = *throws*)

Tries to lock the mutex. Blocks until specified *abs_time* has been reached or the lock is acquired, whichever comes first. On successful lock acquisition returns *true*, otherwise returns *false*. If *abs_time* has already passed, this function behaves like try_lock(). As with try_lock(), this function is allowed to fail spuriously and return *false* even if the mutex was not locked by any other thread at some point before *abs_time*. Prior unlock() operation on the same mutex *synchronizes-with* (as defined in *std::memory_order*) this operation if it returns *true*. If *try_lock_until* is called by a thread that already owns the mutex, the behavior is undefined.

Note: This overload essentially calls

```
bool try_lock_until(
  hpx::chrono::steady_time_point const& abs_time,
  char const* description, error_code& ec = throws);
```

with *description* as `mutex::try_lock_until`.

Parameters

- **abs_time** – time point to block until
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock_until* returns *true* if the lock was acquired successfully, otherwise *false*.

inline bool **try_lock_for**(*hpx*::*chrono*::steady_duration const &rel_time, char const *description, *error_code* &ec = *throws*)

Tries to lock the mutex. Blocks until specified *rel_time* has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns *true*, otherwise returns *false*. If *rel_time* is less or equal *rel_time.zero()*, the function behaves like try_lock(). This function may block for longer than *rel_time* due to scheduling or resource contention delays. As with try_lock(), this function is allowed to fail spuriously and return *false* even if the mutex was not locked by any other thread at some point during *rel_time*. Prior unlock() operation on the same mutex *synchronizes-with* (as defined in *std::memory_order*) this operation if it returns *true*. If *try_lock_for* is called by a thread that already owns the mutex, the behavior is undefined.

Parameters

- **rel_time** – minimum duration to block for
- **description** – Description of the timed_mutex
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock_for* returns *true* if the lock was acquired successfully, otherwise *false*.

inline bool **try_lock_for**(*hpx*::*chrono*::steady_duration const &rel_time, *error_code* &ec = *throws*)

Tries to lock the mutex. Blocks until specified *rel_time* has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns *true*, otherwise returns *false*. If *rel_time* is less or equal *rel_time.zero()*, the function behaves like try_lock(). This function may block for longer than *rel_time* due to scheduling or resource contention delays. As with try_lock(), this function is allowed to fail spuriously and return *false* even if the mutex was not locked by any other thread at some point during *rel_time*. Prior unlock() operation on the same mutex *synchronizes-with* (as defined in *std::memory_order*) this operation if it returns *true*. If *try_lock_for* is called by a thread that already owns the mutex, the behavior is undefined.

Note: This overload essentially calls

```
bool try_lock_for(
    hpx::chrono::steady_duration const& rel_time,
    char const* description, error_code& ec = throws)
```

with *description* as `mutex::try_lock_for`.

Parameters
- **rel_time** – minimum duration to block for
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock_for* returns *true* if the lock was acquired successfully, otherwise *false*.

void **lock**(char const *description, *error_code* &ec = *throws*)

Locks the mutex. If another thread has already locked the mutex, a call to lock will block execution until the lock is acquired. If lock is called by a thread that already owns the mutex, the behavior is undefined: for example, the program may deadlock. hpx::mutex can detect the invalid usage and throws a *std::system_error* with error condition *resource_deadlock_would_occur* instead of deadlocking. Prior unlock() operations on the same mutex synchronize- with (as defined in *std::memory_order*) this operation.

Note: lock() is usually not called directly: *std::unique_lock*, *std::scoped_lock*, and *std::lock_guard* are used to manage exclusive locking.

Parameters
- **description** – Description of the *mutex*
- **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns void *lock* returns *void*.

inline void **lock**(*error_code* &ec = *throws*)

Locks the mutex. If another thread has already locked the mutex, a call to lock will block execution until the lock is acquired. If lock is called by a thread that already owns the mutex, the behavior is undefined: for example, the program may deadlock. hpx::mutex can detect the invalid usage and throws a *std::system_error* with error condition *resource_deadlock_would_occur* instead of deadlocking. Prior unlock() operations on the same mutex synchronize - with(as defined in *std::memory_order*) this operation.

Note: lock() is usually not called directly: *std::unique_lock*, *std::scoped_lock*, and *std::lock_guard* are used to manage exclusive locking. This overload essentially calls `void lock(char const* description, error_code& ec =throws);` with *description* as `mutex::lock`.

Parameters **ec** – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns void *lock* returns *void*.

bool **try_lock**(char const *description, *error_code* &ec = *throws*)

Tries to lock the *mutex*. Returns immediately. On successful lock acquisition returns *true*, otherwise returns *false*. This function is allowed to fail spuriously and return *false* even if the *mutex* is not currently locked by any other thread. If *try_lock* is called by a thread that already owns the *mutex*, the behavior is undefined. Prior unlock() operation on the same mutex synchronizes-with (as defined in *std::memory_order*) this operation if it returns *true*. Note that prior lock() does not synchronize with this operation if it returns *false*.

Parameters

- `description` – Description of the *mutex*
- `ec` – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock* returns *true* on successful lock acquisition, otherwise returns *false*.

inline bool **try_lock**(*error_code* &ec = *throws*)

Tries to lock the *mutex*. Returns immediately. On successful lock acquisition returns *true*, otherwise returns *false*. This function is allowed to fail spuriously and return *false* even if the *mutex* is not currently locked by any other thread. If *try_lock* is called by a thread that already owns the *mutex*, the behavior is undefined. Prior unlock() operation on the same mutex synchronizes-with (as defined in *std::memory_order*) this operation if it returns *true*. Note that prior lock() does not synchronize with this operation if it returns *false*.

Note: This overload essentially calls

```
void try_lock(char const* description,
              error_code& ec = throws);
```

with *description* as `mutex::try_lock`.

Parameters `ec` – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns bool *try_lock* returns *true* on successful lock acquisition, otherwise returns *false*.

void **unlock**(*error_code* &ec = *throws*)

Unlocks the *mutex*. The *mutex* must be locked by the current thread of execution, otherwise, the behavior is undefined. This operation *synchronizes-with* (as defined in *std::memory_order*) any subsequent *lock* operation that obtains ownership of the same *mutex*.

Parameters `ec` – Used to hold error code value originated during the operation. Defaults to *throws* — A special 'throw on error' *error_code*.

Returns *unlock* returns *void*.

namespace **lcos**

namespace **local**

namespace **threads**

Typedefs

using **thread_id_ref_type** = *thread_id_ref*

using **thread_self** = coroutines::detail::coroutine_self

Functions

thread_id **get_self_id**() noexcept

The function *get_self_id* returns the HPX thread id of the current thread (or zero if the current thread is not a HPX thread).

thread_self ***get_self_ptr**() noexcept

The function *get_self_ptr* returns a pointer to the (OS thread specific) self reference to the current HPX thread.

hpx::no_mutex

Defined in header hpx/mutex.hpp[762].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

struct **no_mutex**

#include <no_mutex.hpp> `no_mutex` class can be used in cases where the shared data between multiple threads can be accessed simultaneously without causing inconsistencies.

Public Static Functions

static inline constexpr void **lock**() noexcept

static inline constexpr bool **try_lock**() noexcept

static inline constexpr void **unlock**() noexcept

namespace **lcos**

namespace **local**

[758] https://en.cppreference.com/w/cpp/named_req/Mutex
[759] https://en.cppreference.com/w/cpp/named_req/StandardLayoutType
[760] https://en.cppreference.com/w/cpp/named_req/TimedMutex
[761] https://en.cppreference.com/w/cpp/named_req/StandardLayoutType
[762] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/mutex.hpp

hpx::once_flag, hpx::call_once

Defined in header hpx/mutex.hpp[763].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_ONCE_INIT

namespace **hpx**

Functions

template<typename **F**, typename ...**Args**>
void **call_once**(*once_flag* &flag, *F* &&f, *Args*&&... args)

Executes the Callable object *f* exactly once, even if called concurrently, from several threads.

In detail:

- If, by the time *call_once* is called, flag indicates that *f* was already called, *call_once* returns right away (such a call to *call_once* is known as passive).
- Otherwise, *call_once* invokes `std::forward<Callable>(f)` with the arguments `std::forward<Args>(args)...` (as if by `hpx::invoke`). Unlike the `hpx::thread` constructor or `hpx::async`, the arguments are not moved or copied because they don't need to be transferred to another thread of execution. (such a call to *call_once* is known as active).
  - If that invocation throws an exception, it is propagated to the caller of *call_once*, and the flag is not flipped so that another call will be attempted (such a call to *call_once* is known as exceptional).
  - If that invocation returns normally (such a call to *call_once* is known as returning), the flag is flipped, and all other calls to *call_once* with the same flag are guaranteed to be passive. All active calls on the same flag form a single total order consisting of zero or more exceptional calls, followed by one returning call. The end of each active call synchronizes-with the next active call in that order. The return from the returning call synchronizes-with the returns from all passive calls on the same flag: this means that all concurrent calls to *call_once* are guaranteed to observe any side-effects made by the active call, with no additional synchronization.

Note: If concurrent calls to *call_once* pass different functions *f*, it is unspecified which f will be called. The selected function runs in the same thread as the *call_once* invocation it was passed to. Initialization of function-local statics is guaranteed to occur only once even when called from multiple threads, and may be more efficient than the equivalent code using `hpx::call_once`. The POSIX equivalent of this function is *pthread_once*.

Parameters

- **flag** – an object, for which exactly one function gets executed
- **f** – Callable object to invoke

[763] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/mutex.hpp

- **args** – arguments to pass to the function

Throws *std*::system_error – if any condition prevents calls to *call_once* from executing as specified or any exception thrown by *f*

struct **once_flag**

#include <*once.hpp*> The class *hpx::once_flag* is a helper structure for hpx::call_once. An object of type *hpx::once_flag* that is passed to multiple calls to hpx::call_once allows those calls to coordinate with each other such that only one of the calls will actually run to completion. *hpx::once_flag* is neither copyable nor movable.

Public Functions

HPX_NON_COPYABLE(*once_flag*)

inline **once_flag**() noexcept

Constructs an once_flag object. The internal state is set to indicate that no function has been called yet.

Private Members

std::atomic<long> **status_**

lcos::*local*::*event* **event_**

Friends

template<typename **F**, typename ...**Args**>
friend void **call_once**(*once_flag* &flag, *F* &&f, *Args*&&... args)

Executes the Callable object *f* exactly once, even if called concurrently, from several threads.

In detail:

- If, by the time *call_once* is called, flag indicates that *f* was already called, *call_once* returns right away (such a call to *call_once* is known as passive).
- Otherwise, *call_once* invokes std::forward<Callable>(f) with the arguments std::forward<Args>(args)... (as if by hpx::invoke). Unlike the hpx::thread constructor or hpx::async, the arguments are not moved or copied because they don't need to be transferred to another thread of execution. (such a call to *call_once* is known as active).
  - If that invocation throws an exception, it is propagated to the caller of *call_once*, and the flag is not flipped so that another call will be attempted (such a call to *call_once* is known as exceptional).
  - If that invocation returns normally (such a call to *call_once* is known as returning), the flag is flipped, and all other calls to *call_once* with the same flag are guaranteed to be passive. All active calls on the same flag form a single total order consisting of zero or more exceptional calls, followed by one returning call. The end of each active call synchronizes-with the next active call in that order. The return from the returning call synchronizes-with the returns from all passive calls on the same flag: this means that all concurrent calls to *call_once* are guaranteed to observe any side-effects made by the active call, with no additional synchronization.

Note: If concurrent calls to *call_once* pass different functions *f*, it is unspecified which f will be called. The selected function runs in the same thread as the *call_once* invocation it was passed to. Initialization of function-local statics is guaranteed to occur only once even when called from multiple threads, and may be more efficient than the equivalent code using `hpx::call_once`. The POSIX equivalent of this function is *pthread_once*.

Parameters
- **flag** – an object, for which exactly one function gets executed
- **f** – Callable object to invoke
- **args** – arguments to pass to the function

Throws `std::system_error` – if any condition prevents calls to *call_once* from executing as specified or any exception thrown by *f*

namespace **lcos**

namespace **local**

Functions

```
template<typename F, typename... Args>  HPX_DEPRECATED_V (1, 8,
"hpx::lcos::local::call_once is deprecated,
use hpx::call_once " "instead") void call_once(hpx
```

hpx::recursive_mutex

Defined in header hpx/mutex.hpp[764].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Typedefs

using **recursive_mutex** = detail::recursive_mutex_impl<>

namespace **lcos**

namespace **local**

[764] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/mutex.hpp

hpx::shared_mutex

Defined in header hpx/shared_mutex.hpp[765].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Typedefs

using **shared_mutex** = detail::shared_mutex<>

The *shared_mutex* class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In contrast to other mutex types which facilitate exclusive access, a *shared_mutex* has two levels of access:

- *shared* - several threads can share ownership of the same mutex.
- *exclusive* - only one thread can own the mutex.

If one thread has acquired the exclusive lock (through *lock*, *try_lock*), no other threads can acquire the lock (including the shared). If one thread has acquired the shared lock (through *lock_shared*, *try_lock_shared*), no other thread can acquire the exclusive lock, but can acquire the shared lock. Only when the exclusive lock has not been acquired by any thread, the shared lock can be acquired by multiple threads. Within one thread, only one lock (shared or exclusive) can be acquired at the same time. Shared mutexes are especially useful when shared data can be safely read by any number of threads simultaneously, but a thread may only write the same data when no other thread is reading or writing at the same time. The *shared_mutex* class satisfies all requirements of *SharedMutex* and *StandardLayoutType*.

namespace **lcos**

namespace **local**

hpx/synchronization/sliding_semaphore.hpp

Defined in header hpx/synchronization/sliding_semaphore.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[765] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/shared_mutex.hpp

Typedefs

using **sliding_semaphore** = *sliding_semaphore_var*<>

template<typename **Mutex** = *hpx*::*spinlock*>

class **sliding_semaphore_var**

#include <sliding_semaphore.hpp> A semaphore is a protected variable (an entity storing a value) or abstract data type (an entity grouping several variables that may or may not be numerical) which constitutes the classic method for restricting access to shared resources, such as shared memory, in a multiprogramming environment. Semaphores exist in many variants, though usually the term refers to a counting semaphore, since a binary semaphore is better known as a mutex. A counting semaphore is a counter for a set of available resources, rather than a locked/unlocked flag of a single resource. It was invented by Edsger Dijkstra. Semaphores are the classic solution to preventing race conditions in the dining philosophers problem, although they do not prevent resource deadlocks.

Sliding semaphores can be used for synchronizing multiple threads as well: one thread waiting for several other threads to touch (signal) the semaphore, or several threads waiting for one other thread to touch this semaphore. The difference to a counting semaphore is that a sliding semaphore will not limit the number of threads which are allowed to proceed, but will make sure that the difference between the (arbitrary) number passed to set and wait does not exceed a given threshold.

Public Functions

sliding_semaphore_var(*sliding_semaphore_var* const&) = delete

sliding_semaphore_var &**operator=**(*sliding_semaphore_var* const&) = delete

sliding_semaphore_var(*sliding_semaphore_var*&&) = delete

sliding_semaphore_var &**operator=**(*sliding_semaphore_var*&&) = delete

inline explicit **sliding_semaphore_var**(*std*::int64_t max_difference, *std*::int64_t lower_limit = 0) noexcept

Construct a new sliding semaphore.

Parameters

- **max_difference** – [in] The max difference between the upper limit (as set by *wait()*) and the lower limit (as set by *signal()*) which is allowed without suspending any thread calling *wait()*.
- **lower_limit** – [in] The initial lower limit.

inline void **set_max_difference**(*std*::int64_t max_difference, *std*::int64_t lower_limit = 0) noexcept

Set/Change the difference that will cause the semaphore to trigger

Parameters

- **max_difference** – [in] The max difference between the upper limit (as set by *wait()*) and the lower limit (as set by *signal()*) which is allowed without suspending any thread calling *wait()*.
- **lower_limit** – [in] The initial lower limit.

inline void **wait**(*std*::int64_t upper_limit)

Wait for the semaphore to be signaled.

Parameters **upper_limit** – [in] The new upper limit. The calling thread will be suspended if the difference between this value and the largest lower_limit which was set by *signal()* is larger than the max_difference.

inline bool **try_wait**(*std*::int64_t upper_limit = 1)

Try to wait for the semaphore to be signaled.

Parameters upper_limit – [in] The new upper limit. The calling thread will be suspended if the difference between this value and the largest lower_limit which was set by *signal()* is larger than the max_difference.

Returns The function returns true if the calling thread would not block if it was calling *wait()*.

inline void **signal**(*std*::int64_t lower_limit)

Signal the semaphore.

Parameters lower_limit – [in] The new lower limit. This will update the current lower limit of this semaphore. It will also re-schedule all suspended threads for which their associated upper limit is not larger than the lower limit plus the max_difference.

inline *std*::int64_t **signal_all**()

Private Types

using **mutex_type** = *Mutex*

using **data_type** = *lcos*::*local*::detail::sliding_semaphore_data<*mutex_type*>

Private Members

hpx::intrusive_ptr<*data_type*> **data_**

namespace **lcos**

namespace **local**

hpx::spinlock

Defined in header hpx/mutex.hpp[766].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[766] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/mutex.hpp

Typedefs

using **spinlock** = detail::spinlock<true>

`spinlock` is a type of lock that causes a thread attempting to obtain it to check for its availability while waiting in a loop continuously.

using **spinlock_no_backoff** = detail::spinlock<false>

namespace **lcos**

namespace **local**

hpx::nostopstate, hpx::stop_callback, hpx::stop_source, hpx::stop_token, hpx::nostopstate_t

Defined in header hpx/stop_token.hpp[767].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **Callback**>
stop_callback(*stop_token*, *Callback*) -> *stop_callback*<*Callback*>

The stop_callback class template provides an RAII object type that registers a callback function for an associated hpx::stop_token object, such that the callback function will be invoked when the hpx::stop_token*'s* associated hpx::stop_source is requested to stop. Callback functions registered via stop_callback*'s* constructor are invoked either in the same thread that successfully invokes *request_stop()* for a hpx::stop_source of the stop_callback*'s* associated hpx::stop_token; or if stop has already been requested prior to the constructor's registration, then the callback is invoked in the thread constructing the stop_callback. More than one stop_callback can be created for the same hpx::stop_token, from the same or different threads concurrently. No guarantee is provided for the order in which they will be executed, but they will be invoked synchronously; except for *stop_callback(s)* constructed after stop has already been requested for the hpx::stop_token, as described previously. If an invocation of a callback exits via an exception then hpx::terminate is called. hpx::stop_callback is not *CopyConstructible*, *CopyAssignable*, *MoveConstructible*, nor *MoveAssignable*. The template param Callback type must be both *invocable* and *destructible*. Any return value is ignored.

inline void **swap**(*stop_token* &lhs, *stop_token* &rhs) noexcept

inline void **swap**(*stop_source* &lhs, *stop_source* &rhs) noexcept

[767] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/stop_token.hpp

Variables

constexpr *nostopstate_t* **nostopstate** = {}

This is a constant object instance of *hpx::nostopstate_t* for use in constructing an empty *hpx::stop_source*, as a placeholder value in the non-default constructor.

struct **nostopstate_t**

#include <stop_token.hpp> Unit type intended for use as a placeholder in *hpx::stop_source* non-default constructor, that makes the constructed *hpx::stop_source* empty with no associated stop-state.

Public Functions

explicit **nostopstate_t**() = default

template<typename **Callback**>

class **stop_callback**

class **stop_source**

#include <stop_token.hpp> The stop_source class provides the means to issue a stop request, such as for *hpx::jthread* cancellation. A stop request made for one stop_source object is visible to all *stop_sources* and *hpx::stop_tokens* of the same associated stop-state; any hpx::stop_callback(s) registered for associated hpx::stop_token(s) will be invoked, and any *hpx::condition_variable_any* objects waiting on associated hpx::stop_token(s) will be awoken. Once a stop is requested, it cannot be withdrawn. Additional stop requests have no effect.

Note: For the purposes of *hpx::jthread* cancellation the stop_source object should be retrieved from the hpx::jthread object using *get_stop_source()*; or stop should be requested directly from the *hpx::jthread* object using request_stop(). This will then use the same associated stop-state as that passed into the *hpx::jthread's* invoked function argument (i.e., the function being executed on its thread). For other uses, however, a stop_source can be constructed separately using the default constructor, which creates new stop-state.

Public Functions

inline **stop_source**()

inline explicit **stop_source**(*nostopstate_t*) noexcept

inline **stop_source**(*stop_source* const &rhs) noexcept

stop_source(*stop_source*&&) noexcept = default

inline *stop_source* &**operator=**(*stop_source* const &rhs) noexcept

stop_source &**operator=**(*stop_source*&&) noexcept = default

inline **~stop_source**()

inline void **swap**(*stop_source* &s) noexcept

swaps two *stop_source* objects

inline *stop_token* **get_token**() const noexcept

returns a *stop_token* for the associated stop-state

inline bool **stop_possible**() const noexcept

checks whether associated stop-state can be requested to stop

inline bool **stop_requested**() const noexcept

checks whether the associated stop-state has been requested to stop

inline bool **request_stop**() const noexcept

makes a stop request for the associated stop-state, if any

Private Members

hpx::intrusive_ptr<detail::stop_state> **state_**

Friends

inline friend bool **operator==**(*stop_source* const &lhs, *stop_source* const &rhs) noexcept

inline friend bool **operator!=**(*stop_source* const &lhs, *stop_source* const &rhs) noexcept

class **stop_token**

#include <stop_token.hpp> The stop_token class provides the means to check if a stop request has been made or can be made, for its associated hpx::stop_source object. It is essentially a thread-safe "view" of the associated stop-state. The *stop_token* can also be passed to the constructor of hpx::stop_callback, such that the callback will be invoked if the stop_token*'s* associated hpx::stop_source is requested to stop. And stop_token can be passed to the interruptible waiting functions of hpx::condition_variable_any, to interrupt the condition variable's wait if stop is requested.

Note: A stop_token object is not generally constructed independently, but rather retrieved from a *hpx::jthread* or hpx::stop_source. This makes it share the same associated stop-state as the *hpx::jthread* or hpx::stop_source.

Public Types

template<typename **Callback**>

using **callback_type** = *stop_callback*<*Callback*>

Public Functions

constexpr **stop_token**() noexcept = default

stop_token(*stop_token* const &rhs) = default

stop_token(*stop_token*&&) noexcept = default

stop_token &**operator=**(*stop_token* const &rhs) = default

stop_token &**operator=**(*stop_token*&&) noexcept = default

~stop_token() = default

inline void **swap**(*stop_token* &s) noexcept

swaps two *stop_token* objects

inline bool **stop_requested**() const noexcept

checks whether the associated stop-state has been requested to stop

inline bool **stop_possible**() const noexcept

checks whether associated stop-state can be requested to stop

Private Functions

inline explicit **stop_token**(*hpx*::intrusive_ptr<detail::stop_state> state) noexcept

Private Members

hpx::intrusive_ptr<detail::stop_state> **state_**

Friends

friend class stop_callback

friend class stop_source

inline friend constexpr friend bool operator== (stop_token const &lhs, stop_token const &rhs) noexcept

inline friend constexpr friend bool operator!= (stop_token const &lhs, stop_token const &rhs) noexcept

namespace **experimental**

namespace **p2300_stop_token**

Functions

template<typename **Callback**>
in_place_stop_callback(*in_place_stop_token*, *Callback*) -> *in_place_stop_callback*<*Callback*>

template<typename **Callback**>

class **in_place_stop_callback**

class **in_place_stop_source**

Public Functions

inline **in_place_stop_source**() noexcept

inline **~in_place_stop_source**()

in_place_stop_source(*in_place_stop_source* const&) = delete

in_place_stop_source(*in_place_stop_source*&&) noexcept = delete

in_place_stop_source &**operator=**(*in_place_stop_source* const&) = delete

in_place_stop_source &**operator=**(*in_place_stop_source*&&) noexcept = delete

inline *in_place_stop_token* **get_token**() const noexcept

inline bool **request_stop**() noexcept

inline bool **stop_requested**() const noexcept

inline bool **stop_possible**() const noexcept

Private Functions

inline bool **register_callback**(*hpx*::detail::stop_callback_base *cb) noexcept

inline void **remove_callback**(*hpx*::detail::stop_callback_base *cb) noexcept

Private Members

hpx::detail::stop_state **state_**

Friends

friend class in_place_stop_token

friend class in_place_stop_callback

class **in_place_stop_token**

Public Types

template<typename **Callback**>

using **callback_type** = *in_place_stop_callback*<*Callback*>

Public Functions

inline constexpr **in_place_stop_token**() noexcept

~in_place_stop_token() = default

in_place_stop_token(*in_place_stop_token* const &rhs) noexcept = default

inline **in_place_stop_token**(*in_place_stop_token* &&rhs) noexcept

in_place_stop_token &**operator=**(*in_place_stop_token* const &rhs) noexcept = default

inline *in_place_stop_token* &**operator=**(*in_place_stop_token* &&rhs) noexcept

inline bool **stop_requested**() const noexcept

inline bool **stop_possible**() const noexcept

inline void **swap**(*in_place_stop_token* &rhs) noexcept

Private Functions

inline explicit **in_place_stop_token**(*in_place_stop_source* const *source) noexcept

Private Members

in_place_stop_source const ***source_**

Friends

friend class in_place_stop_source

friend class in_place_stop_callback

inline friend constexpr friend bool operator== (in_place_stop_token const &lhs, in_place_stop_token const &rhs) noexcept

inline friend constexpr friend bool operator!
= (in_place_stop_token const &lhs, in_place_stop_token const &rhs) noexcept

inline friend void **swap**(*in_place_stop_token* &x, *in_place_stop_token* &y) noexcept

struct **never_stop_token**

Public Types

template<typename>

using **callback_type** = *callback_impl*

Public Static Functions

static inline constexpr bool **stop_requested**() noexcept

static inline constexpr bool **stop_possible**() noexcept

Friends

inline friend constexpr friend bool operator== (never_stop_token, never_stop_token) noexcept

inline friend constexpr friend bool operator!= (never_stop_token, never_stop_token) noexcept

struct **callback_impl**

Public Functions

template<typename **Callback**>
inline explicit constexpr **callback_impl**(*never_stop_token*, *Callback*&&) noexcept

tag_invoke

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::is_invocable, hpx::is_invocable_r

Defined in header hpx/type_traits.hpp[768].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[768] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/type_traits.hpp

Variables

template<typename **F**, typename ...**Ts**>

constexpr bool **is_invocable_v** = *is_invocable*<*F*, *Ts*...>::value

template<typename **R**, typename **F**, typename ...**Ts**>

constexpr bool **is_invocable_r_v** = *is_invocable_r*<*R*, *F*, *Ts*...>::value

template<typename **F**, typename ...**Ts**>

constexpr bool **is_nothrow_invocable_v** = *is_nothrow_invocable*<*F*, *Ts*...>::value

template<typename **F**, typename ...**Ts**>

struct **is_invocable** : public *hpx*::detail::is_invocable_impl<*F*&&(*Ts*&&...)>

#include <is_invocable.hpp> Determines whether *F* can be invoked with the arguments *Ts*…. Formally, determines whether

```
INVOKE(std::declval<F>(), std::declval<Ts>()...)
```

is well formed when treated as an unevaluated operand, where *INVOKE* is the operation defined in *Callable*.

F, R and all types in the parameter pack Ts shall each be a complete type, (possibly cv-qualified) void, or an array of unknown bound. Otherwise, the behavior is undefined. If an instantiation of a template above depends, directly or indirectly, on an incomplete type, and that instantiation could yield a different result if that type were hypothetically completed, the behavior is undefined.

template<typename **R**, typename **F**, typename ...**Ts**>

struct **is_invocable_r** : public *hpx*::detail::is_invocable_r_impl<*F*&&(*Ts*&&...), *R*>

#include <is_invocable.hpp> Determines whether *F* can be invoked with the arguments *Ts*… to yield a result that is convertible to *R* and the implicit conversion does not bind a reference to a temporary object (since C++23). If *R* is *cv* void, the result can be any type. Formally, determines whether

```
INVOKE<R>(std::declval<F>(), std::declval<Ts>()...)
```

is well formed when treated as an unevaluated operand, where *INVOKE* is the operation defined in *Callable*. Determines whether *F* can be invoked with the arguments *Ts*…. Formally, determines whether

```
INVOKE(std::declval<F>(), std::declval<Ts>()...)
```

is well formed when treated as an unevaluated operand, where *INVOKE* is the operation defined in *Callable*.

F, R and all types in the parameter pack Ts shall each be a complete type, (possibly cv-qualified) void, or an array of unknown bound. Otherwise, the behavior is undefined. If an instantiation of a template above depends, directly or indirectly, on an incomplete type, and that instantiation could yield a different result if that type were hypothetically completed, the behavior is undefined.

template<typename **F**, typename ...**Ts**>

struct **is_nothrow_invocable** : public *hpx*::detail::is_nothrow_invocable_impl<*F*(*Ts*...), *is_invocable_v*<*F*, *Ts*...>>

thread_pool_util

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/thread_pool_util/thread_pool_suspension_helpers.hpp

Defined in header hpx/thread_pool_util/thread_pool_suspension_helpers.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Functions

hpx::*future*<void> **resume_processing_unit**(*thread_pool_base* &pool, *std*::size_t virt_core)

Resumes the given processing unit. When the processing unit has been resumed the returned future will be ready.

Note: Can only be called from an HPX thread. Use resume_processing_unit_cb or to resume the processing unit from outside HPX. Requires that the pool has threads::policies::enable_elasticity set.

Parameters
- **pool** – [in] The thread pool to resume a processing unit on.
- **virt_core** – [in] The processing unit on the pool to be resumed. The processing units are indexed starting from 0.

Returns A `future<void>` which is ready when the given processing unit has been resumed.

void **resume_processing_unit_cb**(*thread_pool_base* &pool, *hpx*::*function*<void()> callback, *std*::size_t virt_core, *error_code* &ec = *throws*)

Resumes the given processing unit. Takes a callback as a parameter which will be called when the processing unit has been resumed.

Note: Requires that the pool has threads::policies::enable_elasticity set.

Parameters
- **pool** – [in] The thread pool to resume a processing unit on.
- **callback** – [in] Callback which is called when the processing unit has been suspended.
- **virt_core** – [in] The processing unit to resume.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

hpx::*future*<void> **suspend_processing_unit**(*thread_pool_base* &pool, *std*::size_t virt_core)

Suspends the given processing unit. When the processing unit has been suspended the returned future will be ready.

Note: Can only be called from an HPX thread. Use suspend_processing_unit_cb or to suspend the processing unit from outside HPX. Requires that the pool has threads::policies::enable_elasticity set.

Parameters
- **pool** – [in] The thread pool to suspend a processing unit from.
- **virt_core** – [in] The processing unit on the pool to be suspended. The processing units are indexed starting from 0.

Throws `hpx::exception` – if called from outside the HPX runtime.

Returns A `future`<void> which is ready when the given processing unit has been suspended.

void **suspend_processing_unit_cb**(*hpx*::*function*<void()> callback, *thread_pool_base* &pool, *std*::size_t virt_core, *error_code* &ec = *throws*)

Suspends the given processing unit. Takes a callback as a parameter which will be called when the processing unit has been suspended.

Note: Requires that the pool has threads::policies::enable_elasticity set.

Parameters
- **pool** – [in] The thread pool to suspend a processing unit from.
- **callback** – [in] Callback which is called when the processing unit has been suspended.
- **virt_core** – [in] The processing unit to suspend.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

hpx::*future*<void> **resume_pool**(*thread_pool_base* &pool)

Resumes the thread pool. When the all OS threads on the thread pool have been resumed the returned future will be ready.

Note: Can only be called from an HPX thread. Use resume_cb or resume_direct to suspend the pool from outside HPX.

Parameters **pool** – [in] The thread pool to resume.

Throws `hpx::exception` – if called from outside the HPX runtime.

Returns A `future`<void> which is ready when the thread pool has been resumed.

void **resume_pool_cb**(*thread_pool_base* &pool, *hpx*::*function*<void()> callback, *error_code* &ec = *throws*)

Resumes the thread pool. Takes a callback as a parameter which will be called when all OS threads on the thread pool have been resumed.

Parameters
- **pool** – [in] The thread pool to resume.
- **callback** – [in] called when the thread pool has been resumed.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

hpx::*future*<void> **suspend_pool**(*thread_pool_base* &pool)

Suspends the thread pool. When the all OS threads on the thread pool have been suspended the returned future will be ready.

Note: Can only be called from an HPX thread. Use suspend_cb or suspend_direct to suspend the pool from outside HPX. A thread pool cannot be suspended from an HPX thread running on the pool itself.

Parameters **pool** – [in] The thread pool to suspend.

Throws *hpx*::*exception* – if called from outside the HPX runtime.

Returns A future<void> which is ready when the thread pool has been suspended.

void **suspend_pool_cb**(*thread_pool_base* &pool, *hpx*::*function*<void()> callback, *error_code* &ec = *throws*)

Suspends the thread pool. Takes a callback as a parameter which will be called when all OS threads on the thread pool have been suspended.

Note: A thread pool cannot be suspended from an HPX thread running on the pool itself.

Parameters

- **pool** – [in] The thread pool to suspend.
- **callback** – [in] called when the thread pool has been suspended.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Throws *hpx*::*exception* – if called from an HPX thread which is running on the pool itself.

thread_support

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::unlock_guard

Defined in header hpx/mutex.hpp[769].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

template<typename **Mutex**>

class **unlock_guard**

#include <unlock_guard.hpp> The class *unlock_guard* is a mutex wrapper that provides a convenient mechanism for releasing a mutex for the duration of a scoped block.

unlock_guard performs the opposite functionality of lock_guard. When a lock_guard object is created, it attempts to take ownership of the mutex it is given. When control leaves the scope in which the lock_guard object was created, the lock_guard is destructed and the mutex is released. Accordingly, when an *unlock_guard* object is created, it attempts to release the ownership of the mutex it is given. So, when control leaves the scope in which the *unlock_guard* object was created, the *unlock_guard* is destructed and the mutex is owned again. In this way, the mutex is unlocked in the constructor and locked in the destructor, so that one can have an unlocked section within a locked one.

[769] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/mutex.hpp

Public Types

using **mutex_type** = *Mutex*

Public Functions

inline explicit constexpr **unlock_guard**(*Mutex* &m) noexcept

HPX_NON_COPYABLE(*unlock_guard*)

inline **~unlock_guard**()

Private Members

Mutex &**m_**

namespace **util**

Typedefs

using **instead** = *hpx*::*unlock_guard*<Mutex>

threading

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::jthread

Defined in header hpx/thread.hpp[770].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

inline void **swap**(*jthread* &lhs, *jthread* &rhs) noexcept

[770] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/thread.hpp

class **jthread**

#include <jthread.hpp> The class *jthread* represents a single thread of execution. It has the same general behavior as hpx::thread, except that *jthread* automatically rejoins on destruction, and can be cancelled/stopped in certain situations. Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, *hpx::terminate* is called. The top-level function may communicate its return value or an exception to the caller via *hpx::promise* or by modifying shared variables (which may require synchronization, see *hpx::mutex* and *hpx::atomic*) Unlike hpx::thread, the jthread logically holds an internal private member of type *hpx::stop_source*, which maintains a shared stop-state. The *jthread* constructor accepts a function that takes a *hpx::stop_token* as its first argument, which will be passed in by the jthread from its internal *stop_source*. This allows the function to check if stop has been requested during its execution, and return if it has. hpx::jthread objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may be not associated with any *jthread* objects (after detach). No two hpx::jthread objects may represent the same thread of execution; hpx::jthread is not *CopyConstructible* or *CopyAssignable*, although it is *MoveConstructible* and *MoveAssignable*.

Public Types

using **id** = *thread*::*id*

using **native_handle_type** = *thread*::*native_handle_type*

Public Functions

inline **jthread**() noexcept

template<typename **F**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*std*::decay_t<*F*>, *jthread*>>>
inline explicit **jthread**(*F* &&f, *Ts*&&... ts)

inline **~jthread**()

jthread(*jthread* const&) = delete

jthread(*jthread* &&x) noexcept = default

jthread &**operator=**(*jthread* const&) = delete

jthread &**operator=**(*jthread*&&) noexcept = default

moves the jthread object

inline void **swap**(*jthread* &t) noexcept

swaps two jthread objects

inline bool **joinable**() const noexcept

checks whether the thread is joinable, i.e. potentially running in parallel context

inline void **join**()

waits for the thread to finish its execution inline void **detach**()

permits the thread to execute independently from the thread handle

inline *id* **get_id**() const noexcept

returns the id of the thread

inline *native_handle_type* **native_handle**()

returns the underlying implementation-defined thread handle

inline *stop_source* **get_stop_source**() noexcept

returns a stop_source object associated with the shared stop state of the thread

inline *stop_token* **get_stop_token**() const noexcept

returns a stop_token associated with the shared stop state of the thread

inline bool **request_stop**() noexcept

requests execution stop via the shared stop state of the thread

Public Static Functions

static inline unsigned int **hardware_concurrency**()

returns the number of concurrent threads supported by the implementation

Private Members

stop_source **ssource_**

hpx::*thread* **thread_** = {}

Private Static Functions

template<typename **F**, typename ...**Ts**>
static inline void **invoke**(*std*::false_type, *F* &&f, *stop_token*&&, *Ts*&&... ts)

template<typename **F**, typename ...**Ts**>
static inline void **invoke**(*std*::true_type, *F* &&f, *stop_token* &&st, *Ts*&&... ts)

hpx::thread, hpx::this_thread::yield, hpx::this_thread::get_id, hpx::this_thread::sleep_for, hpx::this_thread::sleep_until

Defined in header hpx/thread.hpp[771].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<>

struct `std`::**hash**<::*hpx*::*thread*::*id*>

[771] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/thread.hpp

Public Functions

inline *std*::size_t **operator()**(::*hpx*::*thread*::*id* const &id) const

namespace **hpx**

Typedefs

using **thread_termination_handler_type** = *hpx*::*function*<void(*std*::exception_ptr const &e)>

Functions

void **set_thread_termination_handler**(*thread_termination_handler_type* f)

inline void **swap**(*thread* &x, *thread* &y) noexcept

inline bool **operator==**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

inline bool **operator!=**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

inline bool **operator<**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

inline bool **operator>**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

inline bool **operator<=**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

inline bool **operator>=**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

template<typename **Char**, typename **Traits**>
std::basic_ostream<*Char*, *Traits*> &**operator<<**(*std*::basic_ostream<*Char*, *Traits*> &out, *thread*::*id* const &id)

class **thread**

#include <thread.hpp> The class thread represents a single thread of execution. Threads allow multiple functions to execute concurrently. hreads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, *hpx::terminate* is called. The top-level function may communicate its return value or an exception to the caller via *hpx::promise* or by modifying shared variables (which may require synchronization, see hpx::mutex and hpx::atomic) *hpx::thread* objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may not be associated with any thread objects (after detach). No two *hpx::thread* objects may represent the same thread of execution; hpx::thread is not *CopyConstructible* or *CopyAssignable*, although it is *MoveConstructible* and *MoveAssignable*.

Public Types

using **native_handle_type** = *threads*::thread_id_type

Public Functions

thread() noexcept

template<typename **F**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*std*::decay_t<*F*>, *thread*>>>
inline explicit **thread**(*F* &&f)

template<typename **F**, typename ...**Ts**>
inline explicit **thread**(*F* &&f, *Ts*&&... vs)

template<typename **F**>
inline **thread**(*threads*::*thread_pool_base* *pool, *F* &&f)

template<typename **F**, typename ...**Ts**>
inline **thread**(*threads*::*thread_pool_base* *pool, *F* &&f, *Ts*&&... vs)

~thread()

thread(*thread*&&) noexcept

thread &**operator=**(*thread*&&) noexcept

void **swap**(*thread*&) noexcept
swaps two thread objects

inline bool **joinable**() const noexcept
Checks whether the thread is joinable, i.e. potentially running in parallel context

void **join**()
waits for the thread to finish its execution

inline void **detach**()
permits the thread to execute independently from the thread handle

id **get_id**() const noexcept
returns the id of the thread

inline *native_handle_type* **native_handle**() const
returns the underlying implementation-defined thread handle

void **interrupt**(bool flag = true)

bool **interruption_requested**() const

hpx::*future*<void> **get_future**(*error_code* &ec = *throws*)

std::size_t **get_thread_data**() const

std::size_t **set_thread_data**(*std*::size_t)

Public Static Functions

static unsigned int **hardware_concurrency**() noexcept

returns the number of concurrent threads supported by the implementation

static void **interrupt**(*id*, bool flag = true)

Private Types

using **mutex_type** = *hpx*::*spinlock*

Private Functions

void **terminate**(char const *function, char const *reason) const

inline bool **joinable_locked**() const noexcept

inline void **detach_locked**()

void **start_thread**(*threads*::*thread_pool_base* *pool, *hpx*::*move_only_function*<void()> &&func)

Private Members

mutable *mutex_type* **mtx_**

threads::*thread_id_ref_type* **id_**

Private Static Functions

static *threads*::thread_result_type **thread_function_nullary**(*hpx*::*move_only_function*<void()> const &func)

class **id**

Public Functions

id() noexcept = default

inline explicit **id**(*threads*::thread_id_type const &i) noexcept

inline explicit **id**(*threads*::thread_id_type &&i) noexcept

inline explicit **id**(*threads*::*thread_id_ref_type* const &i) noexcept

inline explicit **id**(*threads*::*thread_id_ref_type* &&i) noexcept

inline *threads*::thread_id_type const &**native_handle**() const noexcept

Private Members

threads::thread_id_type **id_**

Friends

friend class thread

friend bool **operator==**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

friend bool **operator!=**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

friend bool **operator<**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

friend bool **operator>**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

friend bool **operator<=**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

friend bool **operator>=**(*thread*::*id* const &x, *thread*::*id* const &y) noexcept

template<typename **Char**, typename **Traits**>
friend *std*::basic_ostream<*Char*, *Traits*> &**operator<<**(*std*::basic_ostream<*Char*, *Traits*>&, *thread*::*id* const&)

namespace **this_thread**

Functions

thread::*id* **get_id**() noexcept

Returns the id of the current thread.

void **yield**() noexcept

Provides a hint to the implementation to reschedule the execution of threads, allowing other threads to run.

> **Note:** The exact behavior of this function depends on the implementation, in particular on the mechanics of the OS scheduler in use and the state of the system. For example, a first-in-first-out realtime scheduler (SCHED_FIFO in Linux) would suspend the current thread and put it on the back of the queue of the same-priority threads that are ready to run (and if there are no other threads at the same priority, yield has no effect).

void **yield_to**(*thread*::*id*) noexcept

threads::*thread_priority* **get_priority**() noexcept

std::ptrdiff_t **get_stack_size**() noexcept

void **interruption_point**()

bool **interruption_enabled**()

bool **interruption_requested**()

void **interrupt**()

void **sleep_until**(*hpx*::*chrono*::steady_time_point const &abs_time)

Blocks the execution of the current thread until specified *abs_time* has been reached.

It is recommended to use the clock tied to *abs_time*, in which case adjustments of the clock may be taken into account. Thus, the duration of the block might be more or less than `abs_time-Clock::now()` at the time of the call, depending on the direction of the adjustment and whether it is honored by the implementation. The function also may block until after *abs_time* has been reached due to process scheduling or resource contention delays.

Parameters **abs_time** – absolute time to block until

inline void **sleep_for**(*hpx*::*chrono*::steady_duration const &rel_time)

Blocks the execution of the current thread for at least the specified *rel_time*. This function may block for longer than *rel_time* due to scheduling or resource contention delays.

It is recommended to use a steady clock to measure the duration. If an implementation uses a system clock instead, the wait time may also be sensitive to clock adjustments.

Parameters **rel_time** – time duration to sleep

std::size_t **get_thread_data**()

std::size_t **set_thread_data**(*std*::size_t)

class **disable_interruption**

Public Functions

disable_interruption()

~disable_interruption()

Private Functions

disable_interruption(*disable_interruption* const&)

disable_interruption &**operator=**(*disable_interruption* const&)

Private Members

bool **interruption_was_enabled_**

Friends

friend class restore_interruption

class **restore_interruption**

Public Functions

explicit **restore_interruption**(*disable_interruption* &d)

~restore_interruption()

Private Functions

restore_interruption(*restore_interruption* const&)

restore_interruption &**operator=**(*restore_interruption* const&)

Private Members

bool **interruption_was_enabled_**

namespace **std**

template<> id >

Public Functions

inline *std*::size_t **operator()**(::*hpx*::*thread*::*id* const &id) const

threading_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::annotated_function

Defined in header hpx/functional.hpp[772].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

[772] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Functions

template<typename **F**>
constexpr *F* &&**annotated_function**(*F* &&f, char const* = nullptr) noexcept

Returns a function annotated with the given annotation.

Annotating includes setting the thread description per thread id.

Parameters **function** –

template<typename **F**>
constexpr *F* &&**annotated_function**(*F* &&f, *std*::string const&) noexcept

namespace **util**

Functions

template<typename **F**>
constexpr decltype(auto) **annotated_function**(*F* &&f, char const *name = nullptr) noexcept

template<typename **F**>
constexpr decltype(auto) **annotated_function**(*F* &&f, *std*::string const &name) noexcept

hpx/threading_base/print.hpp

Defined in header hpx/threading_base/print.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/threading_base/register_thread.hpp

Defined in header hpx/threading_base/register_thread.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Functions

template<typename **F**>
thread_function_type **make_thread_function**(*F* &&f)

template<typename **F**>
thread_function_type **make_thread_function_nullary**(*F* &&f)

void **register_thread**(*threads*::thread_init_data &data, *threads*::*thread_pool_base* *pool, *threads*::*thread_id_ref_type* &id, *error_code* &ec = *hpx*::*throws*)

Create a new *thread* using the given data.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **data** – [in] The data to use for creating the thread.
- **pool** – [in] The thread pool to use for launching the work.
- **id** – [out] The id of the newly created thread (if applicable)
- **ec** – [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Throws *invalid_status* – if the runtime system has not been started yet.

Returns This function will return the internal id of the newly created HPX-thread.

threads::*thread_id_ref_type* **register_thread**(*threads*::thread_init_data &data, *threads*::*thread_pool_base* *pool, *error_code* &ec = *hpx*::*throws*)

void **register_thread**(*threads*::thread_init_data &data, *threads*::*thread_id_ref_type* &id, *error_code* &ec = *throws*)

Create a new *thread* using the given data on the same thread pool as the calling thread, or on the default thread pool if not on an HPX thread.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **data** – [in] The data to use for creating the thread.
- **id** – [out] The id of the newly created thread (if applicable)
- **ec** – [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Throws *invalid_status* – if the runtime system has not been started yet.

Returns This function will return the internal id of the newly created HPX-thread.

threads::*thread_id_ref_type* **register_thread**(*threads*::thread_init_data &data, *error_code* &ec = *throws*)

thread_id_ref_type **register_work**(*threads*::thread_init_data &data, *threads*::*thread_pool_base* *pool, *error_code* &ec = *hpx*::*throws*)

Create a new work item using the given data.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **data** – [in] The data to use for creating the thread.
- **pool** – [in] The thread pool to use for launching the work.
- **ec** – [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Throws `invalid_status` – if the runtime system has not been started yet.

thread_id_ref_type **register_work**(*threads*::thread_init_data &data, *error_code* &ec = *throws*)

Create a new work item using the given data on the same thread pool as the calling thread, or on the default thread pool if not on an HPX thread.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **data** – [in] The data to use for creating the thread.
- **ec** – [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Throws `invalid_status` – if the runtime system has not been started yet.

hpx::scoped_annotation

Defined in header hpx/functional.hpp[773].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

struct **scoped_annotation**

#include <scoped_annotation.hpp> *scoped_annotation* associates a `name` with a section of code (scope). It can be used to visualize code execution in profiling tools like *Intel VTune*, *Apex Profiler*, etc. That allows analyzing performance to figure out which part(s) of code is (are) responsible for performance degradation, etc.

Public Functions

HPX_NON_COPYABLE(*scoped_annotation*)

inline explicit constexpr **scoped_annotation**(char const*) noexcept

template<typename **F**>
inline explicit constexpr **scoped_annotation**(*F*&&) noexcept

inline **~scoped_annotation**()

namespace **util**

[773] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/functional.hpp

Typedefs

typedef *hpx*::*scoped_annotation* **instead**

hpx/threading_base/thread_data.hpp

Defined in header hpx/threading_base/thread_data.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Functions

constexpr *thread_data* ***get_thread_id_data**(*thread_id_ref_type* const &tid) noexcept

constexpr *thread_data* ***get_thread_id_data**(thread_id_type const &tid) noexcept

class **thread_data** : public thread_data_reference_counting

#include <thread_data.hpp> A *thread* is the representation of a HPX thread. It's a first class object in HPX. In our implementation this is a user level thread running on top of one of the OS threads spawned by the *thread-manager*.

A *thread* encapsulates:
- A thread status word (see the functions *thread::get_state* and *thread::set_state*)
- A function to execute (the thread function)
- A frame (in this implementation this is a block of memory used as the threads stack)
- A block of registers (not implemented yet)

Generally, *threads* are not created or executed directly. All functionality related to the management of *threads* is implemented by the thread-manager.

Public Types

using **spinlock_pool** = *util*::spinlock_pool<*thread_data*>

Public Functions

thread_data(*thread_data* const&) = delete

thread_data(*thread_data*&&) = delete

thread_data &**operator=**(*thread_data* const&) = delete

thread_data &**operator=**(*thread_data*&&) = delete inline thread_state **get_state**(*std*::memory_order const order = *std*::memory_order_acquire) const noexcept

The get_state function queries the state of this thread instance.

Note: This function will be seldom used directly. Most of the time the state of a thread will be retrieved by using the function *threadmanager::get_state*.

Returns This function returns the current state of this thread. It will return one of the values as defined by the *thread_state* enumeration.

inline thread_state **set_state**(*thread_schedule_state* const state, *thread_restart_state* state_ex = *thread_restart_state*::*unknown*, *std*::memory_order const load_order = *std*::memory_order_acquire, *std*::memory_order const exchange_order = *std*::memory_order_acq_rel) const noexcept

The set_state function changes the state of this thread instance.

Note: This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the thread-manager. Moreover, changing the thread state using this function does not change its scheduling status. It only sets the thread's status word. To change the thread's scheduling status *threadmanager::set_state* should be used.

Parameters
- **state** – [in] The new state to be set for the thread.
- **state_ex** – [in]
- **load_order** – [in]
- **exchange_order** – [in]

inline bool **set_state_tagged**(*thread_schedule_state* const newstate, thread_state const &prev_state, thread_state &new_tagged_state, *std*::memory_order exchange_order = *std*::memory_order_acq_rel) const noexcept

inline bool **restore_state**(thread_state const new_state, thread_state const old_state, *std*::memory_order const load_order = *std*::memory_order_relaxed, *std*::memory_order const load_exchange = *std*::memory_order_acq_rel) const noexcept

The restore_state function changes the state of this thread instance depending on its current state. It will change the state atomically only if the current state is still the same as passed as the second parameter. Otherwise it won't touch the thread state of this instance.

Note: This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager. Moreover, changing the thread state using this function does not change its scheduling status. It only sets the thread's status word. To change the thread's scheduling status *threadmanager::set_state* should be used.

Parameters
- **new_state** – [in] The new state to be set for the thread.
- **old_state** – [in] The old state of the thread which still has to be the current state.
- **load_order** – [in]
- **load_exchange** – [in]

Returns This function returns *true* if the state has been changed successfully inline bool **restore_state**(*thread_schedule_state* new_state, *thread_restart_state* const state_ex, thread_state old_state, *std*::memory_order const load_exchange = *std*::memory_order_acq_rel) const noexcept

inline constexpr *thread_priority* **get_priority**() const noexcept

inline void **set_priority**(*thread_priority* priority) noexcept

inline bool **interruption_requested**() const noexcept

inline bool **interruption_enabled**() const noexcept

inline bool **set_interruption_enabled**(bool enable) noexcept

inline void **interrupt**(bool flag = true)

bool **interruption_point**(bool throw_on_interrupt = true)

bool **add_thread_exit_callback**(*function*<void()> const &f)

void **run_thread_exit_callbacks**()

void **free_thread_exit_callbacks**()

inline bool **runs_as_child**(*std*::memory_order mo = *std*::memory_order_acquire) const noexcept

inline constexpr bool **is_stackless**() const noexcept

void **destroy_thread**() override

inline constexpr *policies*::scheduler_base ***get_scheduler_base**() const noexcept

inline constexpr *std*::size_t **get_last_worker_thread_num**() const noexcept

inline void **set_last_worker_thread_num**(*std*::size_t last_worker_thread_num) noexcept

inline constexpr *std*::ptrdiff_t **get_stack_size**() const noexcept

inline *thread_stacksize* **get_stack_size_enum**() const noexcept

template<typename **ThreadQueue**>
inline constexpr *ThreadQueue* &**get_queue**() noexcept

inline coroutine_type::result_type **operator()**(*hpx*::execution_base::this_thread::detail::agent_storage *agent_storage)

Execute the thread function.

Returns This function returns the thread state the thread should be scheduled from this point on. The thread manager will use the returned value to set the thread's scheduling status.

inline coroutine_type::result_type **invoke_directly**()

Directly execute the thread function (inline)

Returns This function returns the thread state the thread should be scheduled from this point on. The thread manager will use the returned value to set the thread's scheduling status.

inline virtual thread_id_type **get_thread_id**() const

inline virtual *std*::size_t **get_thread_phase**() const noexcept

virtual *std*::size_t **get_thread_data**() const = 0 virtual *std*::size_t **set_thread_data**(*std*::size_t data) = 0

virtual void **init**() = 0

virtual void **rebind**(thread_init_data &init_data) = 0

thread_data(thread_init_data &init_data, void *queue, *std*::ptrdiff_t stacksize, bool is_stackless = false, *thread_id_addref* addref = *thread_id_addref*::*yes*)

virtual **~thread_data**() override

virtual void **destroy**() noexcept = 0

Public Static Functions

static inline constexpr *std*::uint64_t **get_component_id**() noexcept

Return the id of the component this thread is running in.

static inline constexpr *threads*::*thread_description* **get_description**() noexcept

static inline constexpr *threads*::*thread_description* **set_description**(*threads*::*thread_description*) noexcept

static inline constexpr *threads*::*thread_description* **get_lco_description**() noexcept

static inline constexpr *threads*::*thread_description* **set_lco_description**(*threads*::*thread_description*) noexcept

static inline constexpr *std*::uint32_t **get_parent_locality_id**() noexcept

Return the locality of the parent thread.

static inline constexpr thread_id_type **get_parent_thread_id**() noexcept

Return the thread id of the parent thread.

static inline constexpr *std*::size_t **get_parent_thread_phase**() noexcept

Return the phase of the parent thread.

static inline constexpr *util*::backtrace const ***get_backtrace**() noexcept

static inline constexpr *util*::backtrace const ***set_backtrace**(*util*::backtrace const*) noexcept

Protected Functions

inline *thread_restart_state* **set_state_ex**(*thread_restart_state* const new_state, *std*::memory_order const load_order = *std*::memory_order_acquire, *std*::memory_order const load_exchange = *std*::memory_order_acq_rel) const noexcept

The set_state function changes the extended state of this thread instance.

Note: This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager.

Parameters

- **new_state** – [in] The new extended state to be set for the thread.
- **load_order** – [in]
- **load_exchange** – [in]

void **rebind_base**(thread_init_data &init_data)

Private Members

mutable *std*::atomic<thread_state> **current_state_**

thread_priority **priority_**

bool **requested_interrupt_**

bool **enabled_interrupt_**

bool **ran_exit_funcs_**

const bool **is_stackless_**

std::atomic<bool> **runs_as_child_**

std::forward_list<*hpx*::*function*<void()>> **exit_funcs_**

policies::scheduler_base ***scheduler_base_**

std::size_t **last_worker_thread_num_**

std::ptrdiff_t **stacksize_**

thread_stacksize **stacksize_enum_**

void ***queue_**

hpx/threading_base/thread_description.hpp

Defined in header hpx/threading_base/thread_description.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Functions

std::ostream &**operator<<**(*std*::ostream&, *thread_description* const&)

std::string **as_string**(*thread_description* const &desc)

threads::*thread_description* **get_thread_description**(thread_id_type const &id, *error_code* &ec = *throws*)

The function get_thread_description is part of the thread related API allows to query the description of one of the threads known to the thread-manager.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The thread id of the thread being queried.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns the description of the thread referenced by the *id* parameter. If the thread is not known to the thread-manager the return value will be the string "<unknown>".

threads::*thread_description* **set_thread_description**(thread_id_type const &id, *threads*::*thread_description* const &desc = *threads*::*thread_description*(), *error_code* &ec = *throws*)

threads::*thread_description* **get_thread_lco_description**(thread_id_type const &id, *error_code* &ec = *throws*)

threads::*thread_description* **set_thread_lco_description**(thread_id_type const &id, *threads*::*thread_description* const &desc = *threads*::*thread_description*(), *error_code* &ec = *throws*)

struct **thread_description**

Public Types

enum **data_type**

Values:

enumerator **data_type_description**

enumerator **data_type_address**

Public Functions

thread_description() noexcept = default

inline constexpr **thread_description**(char const*) noexcept

inline explicit constexpr **thread_description**(*std*::string const&) noexcept

template<typename **F**, typename = *std*::enable_if_t<!*std*::is_same_v<*F*, *thread_description*> && !*traits*::is_action_v<*F*>>>
inline explicit constexpr **thread_description**(*F* const&, char const* = nullptr) noexcept

template<typename **Action**, typename = *std*::enable_if_t<*traits*::is_action_v<*Action*>>>
inline explicit constexpr **thread_description**(*Action*, char const* = nullptr) noexcept

inline explicit constexpr **operator bool**() const noexcept

Public Static Functions

static inline constexpr *data_type* **kind**() noexcept

static inline constexpr char const ***get_description**() noexcept

static inline constexpr *std*::size_t **get_address**() noexcept

static inline constexpr bool **valid**() noexcept

Private Functions

void **init_from_alternative_name**(char const *altname)

namespace **util**

hpx/threading_base/thread_helpers.hpp

Defined in header hpx/threading_base/thread_helpers.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **this_thread**

Functions

threads::*thread_restart_state* **suspend**(*threads*::*thread_schedule_state* state, *threads*::thread_id_type nextid, *threads*::*thread_description* const &description = *threads*::*thread_description*("this_thread::suspend"), *error_code* &ec = *throws*)

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to the thread state passed as the parameter.

Note: Must be called from within a HPX-thread.

Throws `If` – `&ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

inline *threads*::*thread_restart_state* **suspend**(*threads*::*thread_schedule_state* state = *threads*::*thread_schedule_state*::*pending*, *threads*::*thread_description* const &description = *threads*::*thread_description*("this_thread::suspend"), *error_code* &ec = *throws*)

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to the thread state passed as the parameter.

Note: Must be called from within a HPX-thread.

Throws `If` – `&ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

threads::*thread_restart_state* **suspend**(*hpx*::*chrono*::steady_time_point const &abs_time, *threads*::thread_id_type id, *threads*::*thread_description* const &description = *threads*::*thread_description*("this_thread::suspend"), *error_code* &ec = *throws*)

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads at the given time.

Note: Must be called from within a HPX-thread.

Throws `If` – `&ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

inline *threads*::*thread_restart_state* **suspend**(*hpx*::*chrono*::steady_time_point const &abs_time, *threads*::*thread_description* const &description = *threads*::*thread_description*("this_thread::suspend"), *error_code* &ec = *throws*)

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads at the given time.

Note: Must be called from within a HPX-thread.

Throws `If – &ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

inline *threads*::*thread_restart_state* **suspend**(*hpx*::*chrono*::steady_duration const &rel_time, *threads*::*thread_description* const &description = *threads*::*thread_description*("this_thread::suspend"), *error_code* &ec = *throws*)

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads after the given duration.

Note: Must be called from within a HPX-thread.

Throws `If – &ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

inline *threads*::*thread_restart_state* **suspend**(*hpx*::*chrono*::steady_duration const &rel_time, *threads*::thread_id_type const &id, *threads*::*thread_description* const &description = *threads*::*thread_description*("this_thread::suspend"), *error_code* &ec = *throws*)

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads after the given duration.

Note: Must be called from within a HPX-thread.

Throws `If – &ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

inline *threads*::*thread_restart_state* **suspend**(*std*::uint64_t ms, *threads*::*thread_description* const &description = *threads*::*thread_description*("this_thread::suspend"), *error_code* &ec = *throws*)

The function *suspend* will return control to the thread manager (suspends the current thread). It sets the new state of this thread to *suspended* and schedules a wakeup for this threads after the given time (specified in milliseconds).

Note: Must be called from within a HPX-thread.

Throws `If` – `&ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

threads::*thread_pool_base* ***get_pool**(*error_code* &ec = *throws*)

Returns a pointer to the pool that was used to run the current thread

Throws `If` – `&ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

namespace **threads**

Functions

thread_state **set_thread_state**(thread_id_type const &id, *thread_schedule_state* state = *thread_schedule_state*::*pending*, *thread_restart_state* stateex = *thread_restart_state*::*signaled*, *thread_priority* priority = *thread_priority*::*normal*, bool retry_on_active = true, *hpx*::*error_code* &ec = *throws*)

Set the thread state of the *thread* referenced by the thread_id *id*.

Note: If the thread referenced by the parameter *id* is in *thread_state::active* state this function schedules a new thread which will set the state of the thread as soon as its not active anymore. The function returns *thread_state::active* in this case.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The thread id of the thread the state should be modified for.
- **state** – [in] The new state to be set for the thread referenced by the *id* parameter.
- **stateex** – [in] The new extended state to be set for the thread referenced by the *id* parameter.

- **priority** – [in]
- **retry_on_active** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns the previous state of the thread referenced by the *id* parameter. It will return one of the values as defined by the *thread_state* enumeration. If the thread is not known to the thread-manager the return value will be *thread_state::unknown*.

thread_id_ref_type **set_thread_state**(thread_id_type const &id, *hpx*::*chrono*::steady_time_point const &abs_time, *std*::atomic<bool> *started, *thread_schedule_state* state = *thread_schedule_state*::*pending*, *thread_restart_state* stateex = *thread_restart_state*::*timeout*, *thread_priority* priority = *thread_priority*::*normal*, bool retry_on_active = true, *error_code* &ec = *throws*)

Set the thread state of the *thread* referenced by the thread_id *id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (at the given time)

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The thread id of the thread the state should be modified for.
- **abs_time** – [in] Absolute point in time for the new thread to be run
- **started** – [in,out] A helper variable allowing to track the state of the timer helper thread
- **state** – [in] The new state to be set for the thread referenced by the *id* parameter.
- **stateex** – [in] The new extended state to be set for the thread referenced by the *id* parameter.
- **priority** – [in]
- **retry_on_active** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns

inline *thread_id_ref_type* **set_thread_state**(thread_id_type const &id, *hpx*::*chrono*::steady_time_point const &abs_time, *thread_schedule_state* state = *thread_schedule_state*::*pending*, *thread_restart_state* stateex = *thread_restart_state*::*timeout*, *thread_priority* priority = *thread_priority*::*normal*, bool retry_on_active = true, *error_code*& = *throws*)

inline *thread_id_ref_type* **set_thread_state**(thread_id_type const &id, *hpx*::*chrono*::steady_duration const &rel_time, *thread_schedule_state* state = *thread_schedule_state*::*pending*, *thread_restart_state* stateex = *thread_restart_state*::*timeout*, *thread_priority* priority = *thread_priority*::*normal*, bool retry_on_active = true, *error_code* &ec = *throws*)

Set the thread state of the *thread* referenced by the thread_id *id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (after the given duration)

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The thread id of the thread the state should be modified for.
- **rel_time** – [in] Time duration after which the new thread should be run
- **state** – [in] The new state to be set for the thread referenced by the *id* parameter.
- **stateex** – [in] The new extended state to be set for the thread referenced by the *id* parameter.
- **priority** – [in]
- **retry_on_active** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns

thread_state **get_thread_state**(thread_id_type const &id, *error_code* &ec = *throws*) noexcept

The function get_thread_backtrace is part of the thread related API allows to query the currently stored thread back trace (which is captured during thread suspension).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception. The function get_thread_state is part of the thread related API. It queries the state of one of the threads known to the thread-manager.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The thread id of the thread being queried.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.
- **id** – [in] The thread id of the thread the state should be modified for.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns the currently captured stack back trace of the thread referenced by the *id* parameter. If the thread is not known to the thread-manager the return value will be the zero.

Returns This function returns the thread state of the thread referenced by the *id* parameter. If the thread is not known to the thread-manager the return value will be *terminated*.

std::size_t **get_thread_phase**(thread_id_type const &id, *error_code* &ec = *throws*) noexcept

The function get_thread_phase is part of the thread related API. It queries the phase of one of the threads known to the thread-manager.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The thread id of the thread the phase should be modified for.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns the thread phase of the thread referenced by the *id* parameter. If the thread is not known to the thread-manager the return value will be ~0.

bool **get_thread_interruption_enabled**(thread_id_type const &id, *error_code* &ec = *throws*)

Returns whether the given thread can be interrupted at this point.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The thread id of the thread which should be queried.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if the given thread can be interrupted at this point in time. It will return *false* otherwise.

bool **set_thread_interruption_enabled**(thread_id_type const &id, bool enable, *error_code* &ec = *throws*)

Set whether the given thread can be interrupted at this point.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The thread id of the thread which should receive the new value.
- **enable** – [in] This value will determine the new interruption enabled status for the given thread.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns the previous value of whether the given thread could have been interrupted.

bool **get_thread_interruption_requested**(thread_id_type const &id, *error_code* &ec = *throws*)

Returns whether the given thread has been flagged for interruption.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The thread id of the thread which should be queried.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if the given thread was flagged for interruption. It will return *false* otherwise.

void **interrupt_thread**(thread_id_type const &id, bool flag, *error_code* &ec = *throws*)

Flag the given thread for interruption.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The thread id of the thread which should be interrupted.
- **flag** – [in] The flag encodes whether the thread should be interrupted (if it is *true*), or 'uninterrupted' (if it is *false*).
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

inline void **interrupt_thread**(thread_id_type const &id, *error_code* &ec = *throws*)

void **interruption_point**(thread_id_type const &id, *error_code* &ec = *throws*)

Interrupt the current thread at this point if it was canceled. This will throw a thread_interrupted exception, which will cancel the thread.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The thread id of the thread which should be interrupted.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

threads::*thread_priority* **get_thread_priority**(thread_id_type const &id, *error_code* &ec = *throws*) noexcept

Return priority of the given thread

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The thread id of the thread whose priority is queried.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

std::ptrdiff_t **get_stack_size**(thread_id_type const &id, *error_code* &ec = *throws*) noexcept

Return stack size of the given thread

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The thread id of the thread whose priority is queried.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

threads::*thread_pool_base* ***get_pool**(thread_id_type const &id, *error_code* &ec = *throws*)

Returns a pointer to the pool that was used to run the current thread

Throws `If` – `&ec != &throws`, never throws, but will set *ec* to an appropriate value when an error occurs. Otherwise, this function will throw an *hpx::exception* with an error code of *hpx::error::yield_aborted* if it is signaled with *wait_aborted*. If called outside of a HPX-thread, this function will throw an *hpx::exception* with an error code of *hpx::error::null_thread_id*. If this function is called while the thread-manager is not running, it will throw an *hpx::exception* with an error code of *hpx::error::invalid_status*.

hpx::get_worker_thread_num, hpx::get_local_worker_thread_num, hpx::get_local_worker_thread_num, hpx::get_thread_pool_num, hpx::get_thread_pool_num

Defined in header hpx/runtime.hpp[774].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::size_t **get_worker_thread_num**()

Return the number of the current OS-thread running in the runtime instance the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread which executes the current HPX-thread.

Note: The returned value is zero based and its maximum value is smaller than the overall number of OS-threads executed (as returned by *get_os_thread_count()*.

Note: This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

std::size_t **get_worker_thread_num**(*error_code* &ec)

Return the number of the current OS-thread running in the runtime instance the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread which executes the current HPX-thread.

Note: The returned value is zero based and its maximum value is smaller than the overall number of OS-threads executed (as returned by *get_os_thread_count()*. It will return -1 if the current thread is not a known thread or if the runtime is not in running state.

Note: This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters **ec** – [in,out] this represents the error status on exit.

std::size_t **get_local_worker_thread_num**()

Return the number of the current OS-thread running in the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread on the current thread pool which executes the current HPX-thread.

Note: The returned value is zero based and its maximum value is smaller than the number of OS-threads executed on the current thread pool. It will return -1 if the current thread is not a known thread or if the runtime is not in running state.

[774] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

Note: This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

std::size_t **get_local_worker_thread_num**(*error_code* &ec)

Return the number of the current OS-thread running in the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the OS-thread on the current thread pool which executes the current HPX-thread.

Note: The returned value is zero based and its maximum value is smaller than the number of OS-threads executed on the current thread pool. It will return -1 if the current thread is not a known thread or if the runtime is not in running state.

Note: This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters `ec` – [in,out] this represents the error status on exit.

std::size_t **get_thread_pool_num**()

Return the number of the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the thread pool which executes the current HPX-thread.

Note: The returned value is zero based and its maximum value is smaller than the number of thread pools started by the runtime. It will return -1 if the current thread pool is not a known thread pool or if the runtime is not in running state.

Note: This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

std::size_t **get_thread_pool_num**(*error_code* &ec)

Return the number of the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the thread pool which executes the current HPX-thread.

Note: The returned value is zero based and its maximum value is smaller than the number of thread pools started by the runtime. It will return -1 if the current thread pool is not a known thread pool or if the runtime is not in running state.

Note: This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters `ec` – [in,out] this represents the error status on exit.

namespace **threads**

hpx/threading_base/thread_pool_base.hpp

Defined in header hpx/threading_base/thread_pool_base.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Functions

std::ostream &**operator<<**(*std*::ostream &os, *thread_pool_base* const &thread_pool)

class **thread_pool_base**

#include <thread_pool_base.hpp> The base class used to manage a pool of OS threads.

Public Functions

virtual void **suspend_processing_unit_direct**(*std*::size_t virt_core, *error_code* &ec = *throws*) = 0

Suspends the given processing unit. Blocks until the processing unit has been suspended.

Parameters

- **virt_core** – [in] The processing unit on the the pool to be suspended. The processing units are indexed starting from 0.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

virtual void **resume_processing_unit_direct**(*std*::size_t virt_core, *error_code* &ec = *throws*) = 0

Resumes the given processing unit. Blocks until the processing unit has been resumed.

Parameters

- **virt_core** – [in] The processing unit on the the pool to be resumed. The processing units are indexed starting from 0.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

virtual void **resume_direct**(*error_code* &ec = *throws*) = 0

Resumes the thread pool. Blocks until all OS threads on the thread pool have been resumed.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

virtual void **suspend_direct**(*error_code* &ec = *throws*) = 0

Suspends the thread pool. Blocks until all OS threads on the thread pool have been suspended.

Note: A thread pool cannot be suspended from an HPX thread running on the pool itself.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Throws hpx::exception – if called from an HPX thread which is running on the pool itself.

struct **thread_pool_init_parameters**

Public Functions

inline **thread_pool_init_parameters**(std::string const &name, std::size_t index, policies::scheduler_mode mode, std::size_t num_threads, std::size_t thread_offset, hpx::threads::policies::callback_notifier &notifier, hpx::threads::policies::detail::affinity_data const &affinity_data, hpx::threads::detail::network_background_callback_type const &network_background_callback = hpx::threads::detail::network_background_callback_type(), std::size_t max_background_threads = static_cast<std::size_t>(-1), std::size_t max_idle_loop_count = HPX_IDLE_LOOP_COUNT_MAX, std::size_t max_busy_loop_count = HPX_BUSY_LOOP_COUNT_MAX, std::size_t shutdown_check_count = 10)

Public Members

std::string const &**name_**

std::size_t **index_**

policies::scheduler_mode **mode_**

std::size_t **num_threads_**

std::size_t **thread_offset_**

hpx::threads::policies::callback_notifier &**notifier_**

hpx::threads::policies::detail::affinity_data const &**affinity_data_**

hpx::threads::detail::network_background_callback_type const &**network_background_callback_**

std::size_t **max_background_threads_**

std::size_t **max_idle_loop_count_**

std::size_t **max_busy_loop_count_**

std::size_t **shutdown_check_count_**

hpx/threading_base/threading_base_fwd.hpp

Defined in header hpx/threading_base/threading_base_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Functions

thread_data ***get_self_id_data**() noexcept

The function *get_self_id_data* returns the data of the HPX thread id associated with the current thread (or nullptr if the current thread is not a HPX thread).

thread_self &**get_self**()

The function *get_self* returns a reference to the (OS thread specific) self reference to the current HPX thread.

thread_self ***get_self_ptr**() noexcept

The function *get_self_ptr* returns a pointer to the (OS thread specific) self reference to the current HPX thread.

thread_self_impl_type ***get_ctx_ptr**()

The function *get_ctx_ptr* returns a pointer to the internal data associated with each coroutine.

thread_self ***get_self_ptr_checked**(*error_code* &ec = *throws*)

The function *get_self_ptr_checked* returns a pointer to the (OS thread specific) self reference to the current HPX thread.

thread_id_type **get_self_id**() noexcept

The function *get_self_id* returns the HPX thread id of the current thread (or zero if the current thread is not a HPX thread).

thread_id_type **get_outer_self_id**() noexcept

The function *get_outer_self_id* returns the HPX thread id of the current outer thread (or zero if the current thread is not a HPX thread). This usually returns the same as *get_self_id*, except for directly executed threads, in which case this returns the thread id of the outermost HPX thread.

thread_id_type **get_parent_id**() noexcept

The function *get_parent_id* returns the HPX thread id of the current thread's parent (or zero if the current thread is not a HPX thread).

Note: This function will return a meaningful value only if the code was compiled with HPX_HAVE_THREAD_PARENT_REFERENCE being defined.

std::size_t **get_parent_phase**() noexcept

The function *get_parent_phase* returns the HPX phase of the current thread's parent (or zero if the current thread is not a HPX thread).

Note: This function will return a meaningful value only if the code was compiled with HPX_HAVE_THREAD_PARENT_REFERENCE being defined.

std::ptrdiff_t **get_self_stacksize**() noexcept

The function *get_self_stacksize* returns the stack size of the current thread (or zero if the current thread is not a HPX thread).

thread_stacksize **get_self_stacksize_enum**() noexcept

The function *get_self_stacksize_enum* returns the stack size of the /.

std::uint32_t **get_parent_locality_id**() noexcept

The function *get_parent_locality_id* returns the id of the locality of the current thread's parent (or zero if the current thread is not a HPX thread).

Note: This function will return a meaningful value only if the code was compiled with HPX_HAVE_THREAD_PARENT_REFERENCE being defined.

std::uint64_t **get_self_component_id**() noexcept

The function *get_self_component_id* returns the lva of the component the current thread is acting on

Note: This function will return a meaningful value only if the code was compiled with HPX_HAVE_THREAD_TARGET_ADDRESS being defined.

namespace **policies**

threadmanager

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/modules/threadmanager.hpp

Defined in header hpx/modules/threadmanager.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

class **threadmanager**

#include <threadmanager.hpp> The *thread-manager* class is the central instance of management for all (non-depleted) threads

Public Types

using **notification_policy_type** = *threads*::*policies*::callback_notifier

using **pool_type** = *std*::unique_ptr<*thread_pool_base*>

using **pool_vector** = *std*::vector<*pool_type*>

Public Functions

threadmanager(*hpx*::*util*::runtime_configuration &rtcfg_, *notification_policy_type* &notifier, detail::network_background_callback_type const &network_background_callback = detail::network_background_callback_type())

threadmanager(*threadmanager* const&) = delete

threadmanager(*threadmanager*&&) = delete

threadmanager &**operator=**(*threadmanager* const&) = delete

threadmanager &**operator=**(*threadmanager*&&) = delete

~threadmanager()

void **init**() const

void **create_pools**()

void **print_pools**(*std*::ostream&) const

FIXME move to private and add —hpx:printpools cmd line option.

thread_pool_base &**default_pool**() const

thread_pool_base &**get_pool**(*std*::string const &pool_name) const

thread_pool_base &**get_pool**(pool_id_type const &pool_id) const

thread_pool_base &**get_pool**(*std*::size_t thread_index) const

bool **pool_exists**(*std*::string const &pool_name) const

bool **pool_exists**(*std*::size_t pool_index) const

thread_id_ref_type **register_work**(thread_init_data &data, *error_code* &ec = *throws*) const

The function *register_work* adds a new work item to the thread manager. It doesn't immediately create a new *thread*, it just adds the task parameters (function, initial state and description) to the internal management data structures. The thread itself will be created when the number of existing threads drops below the number of threads specified by the constructors max_count parameter.

Parameters

- **data** – [in] The value of this parameter allows to specify a description of the thread to create. This information is used for logging purposes mainly, but might be useful for debugging as well. This parameter is optional and defaults to an empty string.
- **ec** –

void **register_thread**(thread_init_data &data, *thread_id_ref_type* &id, *error_code* &ec = *throws*) const

The function *register_thread* adds a new work item to the thread manager. It creates a new *thread*, adds it to the internal management data structures, and schedules the new thread, if appropriate.

Parameters

- **data** – [in] The value of this parameter allows to specify a description of the thread to create. This information is used for logging purposes mainly, but might be useful for debugging as well. This parameter is optional and defaults to an empty string.
- **id** – [out] This parameter will hold the id of the created thread. This id is guaranteed to be validly initialized before the thread function is executed.
- **ec** –

bool **run**() const

Run the thread manager's work queue. This function instantiates the specified number of OS threads in each pool. All OS threads are started to execute the function *tfunc*.

Returns The function returns *true* if the thread manager has been started successfully, otherwise it returns *false*.

void **stop**(bool blocking = true) const

Forcefully stop the thread-manager.

Parameters **blocking** –

bool **is_busy**() const

bool **is_idle**() const

void **wait**() const

bool **wait_for**(*hpx*::*chrono*::steady_duration const &rel_time) const

void **suspend**() const

void **resume**() const

hpx::state **status**() const

Return whether the thread manager is still running This returns the "minimal state", i.e. the state of the least advanced thread pool

std::int64_t **get_thread_count**(*thread_schedule_state* state = *thread_schedule_state*::*unknown*, *thread_priority* priority = *thread_priority*::*default_*, *std*::size_t num_thread = static_cast<*std*::size_t>(-1), bool reset = false) const

return the number of HPX-threads with the given state

Note: This function locks the internal OS lock in the thread manager

std::int64_t **get_idle_core_count**() const

mask_type **get_idle_core_mask**() const

std::int64_t **get_background_thread_count**() const

bool **enumerate_threads**(*hpx*::*function*<bool(thread_id_type)> const &f, *thread_schedule_state* state = *thread_schedule_state*::*unknown*) const void **abort_all_suspended_threads**() const

bool **cleanup_terminated**(bool delete_all) const

std::size_t **get_os_thread_count**() const

Return the number of OS threads running in this thread-manager.

This function will return correct results only if the thread-manager is running.

std::thread &**get_os_thread_handle**(*std*::size_t num_thread) const

void **report_error**(*std*::size_t num_thread, *std*::exception_ptr const &e) const

API functions forwarding to notification policy.

This notifies the thread manager that the passed exception has been raised. The exception will be routed through the notifier and the scheduler (which will result in it being passed to the runtime object, which in turn will report it to the console, etc.).

mask_type **get_used_processing_units**() const

Returns the mask identifying all processing units used by this thread manager.

hwloc_bitmap_ptr **get_pool_numa_bitmap**(*std*::string const &pool_name) const

void **set_scheduler_mode**(*threads*::*policies*::scheduler_mode mode) const noexcept

void **add_scheduler_mode**(*threads*::*policies*::scheduler_mode mode) const noexcept

void **add_remove_scheduler_mode**(*threads*::*policies*::scheduler_mode to_add_mode, *threads*::*policies*::scheduler_mode to_remove_mode) const noexcept

void **remove_scheduler_mode**(*threads*::*policies*::scheduler_mode mode) const noexcept

void **reset_thread_distribution**() const noexcept

std::int64_t **get_queue_length**(bool reset) const

std::int64_t **get_cumulative_duration**(bool reset) const

std::int64_t **get_thread_count_unknown**(bool reset) const

std::int64_t **get_thread_count_active**(bool reset) const

std::int64_t **get_thread_count_pending**(bool reset) const

std::int64_t **get_thread_count_suspended**(bool reset) const

std::int64_t **get_thread_count_terminated**(bool reset) const

std::int64_t **get_thread_count_staged**(bool reset) const

Public Static Functions

static void **init_tss**(*std*::size_t global_thread_num)

static void **deinit_tss**()

Private Types

using **mutex_type** = *std*::mutex

Private Functions

policies::thread_queue_init_parameters **get_init_parameters**() const

void **create_scheduler_user_defined**(*hpx*::*resource*::scheduler_function const&, *thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&)

void **create_scheduler_local**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_local_priority_fifo**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_local_priority_lifo**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_static**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_static_priority**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_abp_priority_fifo**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_abp_priority_lifo**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_shared_priority**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_local_workrequesting_fifo**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t) void **create_scheduler_local_workrequesting_lifo**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

void **create_scheduler_local_workrequesting_mc**(*thread_pool_init_parameters* const&, *policies*::thread_queue_init_parameters const&, *std*::size_t)

Private Members

mutable *mutex_type* **mtx_**

hpx::*util*::runtime_configuration &**rtcfg_**

std::vector<pool_id_type> **threads_lookup_**

pool_vector **pools_**

notification_policy_type &**notifier_**

detail::network_background_callback_type **network_background_callback_**

timed_execution

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/timed_execution/timed_execution.hpp

Defined in header hpx/timed_execution/timed_execution.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

hpx/timed_execution/timed_execution_fwd.hpp

Defined in header hpx/timed_execution/timed_execution_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Variables

hpx::*parallel*::*execution*::*post_at_t* **post_at**

hpx::*parallel*::*execution*::*post_after_t* **post_after**

hpx::*parallel*::*execution*::*async_execute_at_t* **async_execute_at**

hpx::*parallel*::*execution*::*async_execute_after_t* **async_execute_after**

hpx::*parallel*::*execution*::*sync_execute_at_t* **sync_execute_at**

hpx::*parallel*::*execution*::*sync_execute_after_t* **sync_execute_after**

struct **async_execute_after_t** : public *hpx*::*functional*::detail::tag_fallback<*async_execute_after_t*>

#include <timed_execution_fwd.hpp> Customization point of asynchronous execution agent creation supporting timed execution.

This asynchronously creates a single function invocation f() using the associated executor at the given point in time.

Note: This calls exec.async_execute_after(rel_time, f, ts…), if available, otherwise it emulates timed scheduling by delaying calling execution::async_execute() on the underlying non-time-scheduled execution agent.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param rel_time [in] The duration of time after which the given function should be scheduled to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts… [in] Additional arguments to use to invoke *f*.
Return f(ts…)'s result through a future

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*async_execute_after_t*, *Executor* &&exec, *hpx*::*chrono*::steady_duration const &rel_time, *F* &&f, *Ts*&&... ts)

struct **async_execute_at_t** : public *hpx*::*functional*::detail::tag_fallback<*async_execute_at_t*>

#include <timed_execution_fwd.hpp> Customization point of asynchronous execution agent creation supporting timed execution.

This asynchronously creates a single function invocation f() using the associated executor at the given point in time.

Note: This calls exec.async_execute_at(abs_time, f, ts…), if available, otherwise it emulates timed scheduling by delaying calling execution::async_execute() on the underlying non-time-scheduled execution agent.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param abs_time [in] The point in time the given function should be scheduled at to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts… [in] Additional arguments to use to invoke *f*.
Return f(ts…)'s result through a future

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*async_execute_at_t*, *Executor* &&exec, *hpx*::*chrono*::steady_time_point const &abs_time, *F* &&f, *Ts*&&... ts)

struct **post_after_t** : public *hpx*::*functional*::detail::tag_fallback<*post_after_t*>

#include <timed_execution_fwd.hpp> Customization point of asynchronous fire & forget execution agent creation supporting timed execution.

This asynchronously (fire & forget) creates a single function invocation f() using the associated executor at the given point in time.

Note: This calls exec.post_after(rel_time, f, ts…), if available, otherwise it emulates timed scheduling by delaying calling execution::post() on the underlying non-time-scheduled execution agent.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param rel_time [in] The duration of time after which the given function should be scheduled to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts… [in] Additional arguments to use to invoke *f*.

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*post_after_t*, *Executor* &&exec, *hpx*::*chrono*::steady_duration const &rel_time, *F* &&f, *Ts*&&... ts)

struct **post_at_t** : public *hpx*::*functional*::detail::tag_fallback<*post_at_t*>

#include <timed_execution_fwd.hpp> Customization point of asynchronous fire & forget execution agent creation supporting timed execution.

This asynchronously (fire & forget) creates a single function invocation f() using the associated executor at the given point in time.

Note: This calls exec.post_at(abs_time, f, ts…), if available, otherwise it emulates timed scheduling by delaying calling execution::post() on the underlying non-time-scheduled execution agent.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param abs_time [in] The point in time the given function should be scheduled at to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts… [in] Additional arguments to use to invoke *f*.

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*post_at_t*, *Executor* &&exec, *hpx*::*chrono*::steady_time_point const &abs_time, *F* &&f, *Ts*&&... ts)

struct **sync_execute_after_t** : public *hpx*::*functional*::detail::tag_fallback<*sync_execute_after_t*>

#include <timed_execution_fwd.hpp> Customization point of synchronous execution agent creation supporting timed execution.

This synchronously creates a single function invocation f() using the associated executor at the given point in time.

Note: This calls exec.sync_execute_after(rel_time, f, ts…), if available, otherwise it emulates timed scheduling by delaying calling execution::sync_execute() on the underlying non-time-scheduled execution agent.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param rel_time [in] The duration of time after which the given function should be scheduled to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts… [in] Additional arguments to use to invoke *f*.
Return f(ts…)'s result

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*sync_execute_after_t*, *Executor* &&exec, *hpx*::*chrono*::steady_duration const &rel_time, *F* &&f, *Ts*&&... ts)

struct **sync_execute_at_t** : public *hpx*::*functional*::detail::tag_fallback<*sync_execute_at_t*>

#include <timed_execution_fwd.hpp> Customization point of synchronous execution agent creation supporting timed execution.

This synchronously creates a single function invocation f() using the associated executor at the given point in time.

Note: This calls exec.sync_execute_at(abs_time, f, ts…), if available, otherwise it emulates timed scheduling by delaying calling execution::sync_execute() on the underlying non-time-scheduled execution agent.

Param exec [in] The executor object to use for scheduling of the function *f*.
Param abs_time [in] The point in time the given function should be scheduled at to run.
Param f [in] The function which will be scheduled using the given executor.
Param ts… [in] Additional arguments to use to invoke *f*.
Return f(ts…)'s result

Private Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
inline decltype(auto) friend **tag_fallback_invoke**(*sync_execute_at_t*, *Executor* &&exec, *hpx*::*chrono*::steady_time_point const &abs_time, *F* &&f, *Ts*&&... ts)

template<typename **BaseExecutor**>

struct **timed_executor**

hpx/timed_execution/timed_executors.hpp

Defined in header hpx/timed_execution/timed_executors.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Typedefs

using **sequenced_timed_executor** = *timed_executor*<*hpx*::*execution*::*sequenced_executor*>

using **parallel_timed_executor** = *timed_executor*<*hpx*::*execution*::*parallel_executor*>

template<typename **BaseExecutor**>

struct **timed_executor**

hpx/timed_execution/traits/is_timed_executor.hpp

Defined in header hpx/timed_execution/traits/is_timed_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Typedefs

template<typename **T**>

using **is_timed_executor_t** = typename *is_timed_executor*<*T*>::type

Variables

template<typename **T**>

constexpr bool **is_timed_executor_v** = *is_timed_executor*<*T*>::value

template<typename **T**>

struct **is_timed_executor** : public detail::is_timed_executor<*std*::decay_t<*T*>>

namespace **traits**

template<typename **Executor**, typename **Enable** = void>

struct **is_timed_executor** : public *hpx*::*parallel*::*execution*::*is_timed_executor*<*Executor*>

timing

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::chrono::high_resolution_clock

Defined in header hpx/chrono.hpp[775].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **chrono**

struct **high_resolution_clock**

#include <high_resolution_clock.hpp> Class `hpx::chrono::high_resolution_clock` represents the clock with the smallest tick period provided by the implementation. It may be an alias of `std::chrono::system_clock` or `std::chrono::steady_clock`, or a third, independent clock. `hpx::chrono::high_resolution_clock` meets the requirements of *TrivialClock*.

Public Static Functions

static inline *std*::uint64_t **now**() noexcept

returns a `std::chrono::time_point` representing the current value of the clock

static inline constexpr std::uint64_t() min () noexcept

static inline constexpr std::uint64_t() max () noexcept

hpx::chrono::high_resolution_timer

Defined in header hpx/chrono.hpp[776].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **chrono**

class **high_resolution_timer**

#include <high_resolution_timer.hpp> *high_resolution_timer* is a timer object which measures the elapsed time

[775] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/chrono.hpp

[776] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/core/include_local/include/hpx/chrono.hpp

Public Types

enum class **init**

Values:

enumerator **no_init**

Public Functions

inline **high_resolution_timer**() noexcept

inline explicit constexpr **high_resolution_timer**(*init*) noexcept

inline explicit constexpr **high_resolution_timer**(double t) noexcept

inline void **restart**() noexcept

restarts the timer

inline double **elapsed**() const noexcept

returns the elapsed time in seconds

inline *std*::int64_t **elapsed_microseconds**() const noexcept

returns the elapsed time in microseconds

inline *std*::int64_t **elapsed_nanoseconds**() const noexcept

returns the elapsed time in nanoseconds

Public Static Functions

static inline double **now**() noexcept

returns the current time

static inline constexpr double **elapsed_max**() noexcept

returns the estimated maximum value for `elapsed()`

static inline constexpr double **elapsed_min**() noexcept

returns the estimated minimum value for `elapsed()`

Protected Static Functions

static inline *std*::uint64_t **take_time_stamp**() noexcept

Private Members

std::uint64_t **start_time_**

topology

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/topology/cpu_mask.hpp

Defined in header hpx/topology/cpu_mask.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

hpx/topology/topology.hpp

Defined in header hpx/topology/topology.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **threads**

Typedefs

using **hwloc_bitmap_ptr** = *std*::shared_ptr<*hpx_hwloc_bitmap_wrapper*>

Enums

enum **hpx_hwloc_membind_policy**

Please see hwloc documentation for the corresponding enums HWLOC_MEMBIND_XXX.

Values:

enumerator **membind_default**

enumerator **membind_firsttouch**

enumerator **membind_bind**

enumerator **membind_interleave**

enumerator **membind_replicate**

enumerator **membind_nexttouch**

enumerator **membind_mixed**

enumerator **membind_user**

Functions

topology &**create_topology**()

inline *std*::size_t **get_memory_page_size**()

struct **hpx_hwloc_bitmap_wrapper**

Public Functions

HPX_NON_COPYABLE(*hpx_hwloc_bitmap_wrapper*)

inline **hpx_hwloc_bitmap_wrapper**() noexcept

inline explicit **hpx_hwloc_bitmap_wrapper**(void *bmp) noexcept

inline **~hpx_hwloc_bitmap_wrapper**()

inline void **reset**(hwloc_bitmap_t bmp) noexcept

inline explicit constexpr **operator bool**() const noexcept

inline hwloc_bitmap_t **get_bmp**() const noexcept

Private Members

hwloc_bitmap_t **bmp_**

Friends

friend *std*::ostream &**operator<<**(*std*::ostream &os, *hpx_hwloc_bitmap_wrapper* const *bmp)

struct **topology**

Public Functions

topology()

topology(*topology* const&) = delete

topology(*topology*&&) = delete

topology &**operator=**(*topology* const&) = delete

topology &**operator=**(*topology*&&) = delete

~topology()

inline *std*::size_t **get_socket_number**(*std*::size_t num_thread, [[maybe_unused]] *error_code* &ec = *throws*) const noexcept

Return the Socket number of the processing unit the given thread is running on.

Parameters

- **num_thread** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

inline *std*::size_t **get_numa_node_number**(*std*::size_t num_thread, [[maybe_unused]] *error_code* &ec = *throws*) const noexcept

Return the NUMA node number of the processing unit the given thread is running on.

Parameters

- **num_thread** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_cref_type **get_machine_affinity_mask**(*error_code* &ec = *throws*) const noexcept

Return a bit mask where each set bit corresponds to a processing unit available to the application.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_type **get_service_affinity_mask**(mask_cref_type used_processing_units, *error_code* &ec = *throws*) const

Return a bit mask where each set bit corresponds to a processing unit available to the service threads in the application.

Parameters

- **used_processing_units** – [in] This is the mask of processing units which are not available for service threads.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_cref_type **get_socket_affinity_mask**(*std*::size_t num_thread, *error_code* &ec = *throws*) const

Return a bit mask where each set bit corresponds to a processing unit available to the given thread inside the socket it is running on.

Parameters

- **num_thread** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_cref_type **get_numa_node_affinity_mask**(*std*::size_t num_thread, *error_code* &ec = *throws*) const

Return a bit mask where each set bit corresponds to a processing unit available to the given thread inside the NUMA domain it is running on.

Parameters
- **num_thread** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_cref_type **get_core_affinity_mask**(*std*::size_t num_thread, *error_code* &ec = *throws*) const

Return a bit mask where each set bit corresponds to a processing unit available to the given thread inside the core it is running on.

Parameters
- **num_thread** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_cref_type **get_thread_affinity_mask**(*std*::size_t num_thread, *error_code* &ec = *throws*) const

Return a bit mask where each set bit corresponds to a processing unit available to the given thread.

Parameters
- **num_thread** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

void **set_thread_affinity_mask**(mask_cref_type mask, *error_code* &ec = *throws*) const

Use the given bit mask to set the affinity of the given thread. Each set bit corresponds to a processing unit the thread will be allowed to run on.

Note: Use this function on systems where the affinity must be set from inside the thread itself.

Parameters
- **mask** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_type **get_thread_affinity_mask_from_lva**(void const *lva, *error_code* &ec = *throws*) const

Return a bit mask where each set bit corresponds to a processing unit co-located with the memory the given address is currently allocated on.

Parameters
- **lva** – [in]
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

void **print_affinity_mask**(*std*::ostream &os, *std*::size_t num_thread, mask_cref_type m, *std*::string const &pool_name) const

Prints the given mask *m* to os in a human readable form.

bool **reduce_thread_priority**(*error_code* &ec = *throws*) const

Reduce thread priority of the current thread.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

std::size_t **get_number_of_sockets**() const

Return the number of available NUMA domains.

std::size_t **get_number_of_numa_nodes**() const

Return the number of available NUMA domains.

std::size_t **get_number_of_cores**() const

Return the number of available cores.

std::size_t **get_number_of_pus**() const noexcept

Return the number of available hardware processing units.

std::size_t **get_number_of_numa_node_cores**(*std*::size_t numa) const

Return number of cores in given numa domain.

std::size_t **get_number_of_numa_node_pus**(*std*::size_t numa) const

Return number of processing units in a given numa domain.

std::size_t **get_number_of_socket_pus**(*std*::size_t socket) const

Return number of processing units in a given socket.

std::size_t **get_number_of_core_pus**(*std*::size_t core) const

Return number of processing units in given core.

std::size_t **get_number_of_socket_cores**(*std*::size_t socket) const

Return number of cores units in given socket.

inline *std*::size_t **get_core_number**(*std*::size_t num_thread, *error_code*& = *throws*) const

std::size_t **get_pu_number**(*std*::size_t num_core, *std*::size_t num_pu, *error_code* &ec = *throws*) const

std::size_t **get_cache_size**(mask_cref_type mask, int level) const

Return the size of the cache associated with the given mask.

mask_type **get_cpubind_mask**(*error_code* &ec = *throws*) const

mask_type **get_cpubind_mask**(*std*::thread &handle, *error_code* &ec = *throws*) const

hwloc_bitmap_ptr **cpuset_to_nodeset**(mask_cref_type mask) const

convert a cpu mask into a numa node mask in hwloc bitmap form

void **write_to_log**() const

void ***allocate**(*std*::size_t len) const

This is equivalent to malloc(), except that it tries to allocate page-aligned memory from the OS.

void ***allocate_membind**(*std*::size_t len, const *hwloc_bitmap_ptr* &bitmap, *hpx_hwloc_membind_policy* policy, int flags) const

allocate memory with binding to a numa node set as specified by the policy and flags (see hwloc docs)

threads::mask_type **get_area_membind_nodeset**(void const *addr, *std*::size_t len) const

bool **set_area_membind_nodeset**(void const *addr, *std*::size_t len, void *nodeset) const

int **get_numa_domain**(void const *addr) const void **deallocate**(void *addr, *std*::size_t len) const noexcept

Free memory that was previously allocated by allocate.

void **print_hwloc**(*std*::ostream&) const

mask_type **init_socket_affinity_mask_from_socket**(*std*::size_t num_socket) const

mask_type **init_numa_node_affinity_mask_from_numa_node**(*std*::size_t num_numa_node) const

mask_type **init_core_affinity_mask_from_core**(*std*::size_t num_core, mask_cref_type default_mask = *empty_mask*) const

mask_type **init_thread_affinity_mask**(*std*::size_t num_thread) const

mask_type **init_thread_affinity_mask**(*std*::size_t num_core, *std*::size_t num_pu) const

hwloc_bitmap_t **mask_to_bitmap**(mask_cref_type mask, hwloc_obj_type_t htype) const

mask_type **bitmap_to_mask**(hwloc_bitmap_t bitmap, hwloc_obj_type_t htype) const

Public Static Functions

static void **print_vector**(*std*::ostream &os, *std*::vector<*std*::size_t> const &v)

static void **print_mask_vector**(*std*::ostream &os, *std*::vector<mask_type> const &v)

Private Types

using **mutex_type** = *hpx*::*util*::spinlock

Private Functions

std::size_t **init_node_number**(*std*::size_t num_thread, hwloc_obj_type_t type) const

inline *std*::size_t **init_socket_number**(*std*::size_t num_thread) const

std::size_t **init_numa_node_number**(*std*::size_t num_thread) const

inline *std*::size_t **init_core_number**(*std*::size_t num_thread) const

void **extract_node_mask**(hwloc_obj_t parent, mask_type &mask) const

std::size_t **get_number_of_core_pus_locked**(*std*::size_t core) const

std::size_t **extract_node_count**(hwloc_obj_t parent, hwloc_obj_type_t type, *std*::size_t count) const

std::size_t **extract_node_count_locked**(hwloc_obj_t parent, hwloc_obj_type_t type, *std*::size_t count) const

mask_type **init_machine_affinity_mask**() const

inline mask_type **init_socket_affinity_mask**(*std*::size_t num_thread) const inline mask_type **init_numa_node_affinity_mask**(*std*::size_t num_thread) const

inline mask_type **init_core_affinity_mask**(*std*::size_t num_thread) const

void **init_num_of_pus**()

hwloc_obj_t **get_pu_obj**(*std*::size_t num_pu) const

Private Members

hwloc_topology_t **topo** = nullptr

std::size_t **num_of_pus_** = 0

bool **use_pus_as_cores_** = false

mutable *mutex_type* **topo_mtx**

std::vector<*std*::size_t> **socket_numbers_**

std::vector<*std*::size_t> **numa_node_numbers_**

std::vector<*std*::size_t> **core_numbers_**

mask_type **machine_affinity_mask_** = mask_type()

std::vector<mask_type> **socket_affinity_masks_**

std::vector<mask_type> **numa_node_affinity_masks_**

std::vector<mask_type> **core_affinity_masks_**

std::vector<mask_type> **thread_affinity_masks_**

Private Static Attributes

static mask_type **empty_mask**

static *std*::size_t **memory_page_size_**

static constexpr *std*::size_t **pu_offset** = 0

static constexpr *std*::size_t **core_offset** = 0

Friends

friend *std*::size_t **get_memory_page_size**()

util

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/util/insert_checked.hpp

Defined in header hpx/util/insert_checked.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

Functions

template<typename **Iterator**>
constexpr bool **insert_checked**(*std*::pair<*Iterator*, bool> const &r) noexcept

Helper function for writing predicates that test whether an std::map insertion succeeded. This inline template function negates the need to explicitly write the sometimes lengthy std::pair<Iterator, bool> type.

Parameters **r** – [in] The return value of a std::map insert operation.

Returns This function returns **r.second**.

template<typename **Iterator**>
bool **insert_checked**(*std*::pair<*Iterator*, bool> const &r, *Iterator* &it)

Helper function for writing predicates that test whether an std::map insertion succeeded. This inline template function negates the need to explicitly write the sometimes lengthy std::pair<Iterator, bool> type.

Parameters

- **r** – [in] The return value of a std::map insert operation.
- **r** – [out] A reference to an Iterator, which is set to **r.first**.
- **it** – [out] on exit, will hold the iterator referring to the inserted element

Returns This function returns **r.second**.

hpx/util/sed_transform.hpp

Defined in header hpx/util/sed_transform.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

Functions

bool **parse_sed_expression**(*std*::string const &input, *std*::string &search, *std*::string &replace)

Parse a sed command.

Note: Currently, only supports search and replace syntax (s/search/replace/)

Parameters
- **input** – [in] The content to parse.
- **search** – [out] If the parsing is successful, this string is set to the search expression.
- **replace** – [out] If the parsing is successful, this string is set to the replace expression.

Returns *true* if the parsing was successful, false otherwise.

struct **sed_transform**

#include <sed_transform.hpp> An unary function object which applies a sed command to its subject and returns the resulting string.

Note: Currently, only supports search and replace syntax (s/search/replace/)

Public Functions

sed_transform(*std*::string const &search, *std*::string replace)

explicit **sed_transform**(*std*::string const &expression)

std::string **operator()**(*std*::string const &input) const

inline explicit **operator bool**() const noexcept

inline bool **operator!**() const noexcept

Private Members

std::shared_ptr<command> **command_**

actions

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/actions/action_support.hpp

Defined in header hpx/actions/action_support.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/actions/actions_fwd.hpp

Defined in header hpx/actions/actions_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/actions/base_action.hpp

Defined in header hpx/actions/base_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/actions/transfer_action.hpp

Defined in header hpx/actions/transfer_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/actions/transfer_base_action.hpp

Defined in header hpx/actions/transfer_base_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

actions_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/actions_base/actions_base_fwd.hpp

Defined in header hpx/actions_base/actions_base_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **actions**

hpx/actions_base/actions_base_support.hpp

Defined in header hpx/actions_base/actions_base_support.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **actions**

hpx/actions_base/basic_action.hpp

Defined in header hpx/actions_base/basic_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_ACTION_DECLARATION(...)

Declare the necessary component action boilerplate code.

The macro *HPX_REGISTER_ACTION_DECLARATION* can be used to declare all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to declare the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as ‘<’, ‘>’, or ‘:’.

```
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::component_base<server>
    {
        void print_greeting ()
        {
            hpx::cout << "Hey, how are you?\n" << std::flush;
        }

        // Component actions need to be declared, this also defines the
        // type 'print_greeting_action' representing the action.
        HPX_DEFINE_COMPONENT_ACTION(server,
            print_greeting, print_greeting_action);
    };
}

// Declare boilerplate code required for each of the component actions.
HPX_REGISTER_ACTION_DECLARATION(app::server::print_greeting_action)
```

Example:

Note: This macro has to be used once for each of the component actions defined using one of the *HPX_DEFINE_COMPONENT_ACTION* macros. It has to be visible in all translation units using the action, thus it is recommended to place it into the header file defining the component.

`HPX_REGISTER_ACTION_DECLARATION_`(...)

`HPX_REGISTER_ACTION_DECLARATION_1`(action)

`HPX_REGISTER_ACTION`(...)

Define the necessary component action boilerplate code.

The macro *HPX_REGISTER_ACTION* can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as ‘<’, ‘>’, or ‘:’.

Note: This macro has to be used once for each of the component actions defined using one of the *HPX_DEFINE_COMPONENT_ACTION* or *HPX_DEFINE_PLAIN_ACTION* macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note: Only one of the forms of this macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID* should be used for a particular action, never both.

`HPX_REGISTER_ACTION_ID`(action, actionname, actionid)

Define the necessary component action boilerplate code and assign a predefined unique id to the action.

The macro *HPX_REGISTER_ACTION* can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

The parameter *actionname* specifies an unique name of the action to be used for serialization purposes. The second parameter has to be usable as a plain (non-qualified) C++ identifier, it should not contain special characters which cannot be part of a C++ identifier, such as ‘<’, ‘>’, or ‘:’.

The parameter *actionid* specifies an unique integer value which will be used to represent the action during serialization.

Note: This macro has to be used once for each of the component actions defined using one of the *HPX_DEFINE_COMPONENT_ACTION* or global actions *HPX_DEFINE_PLAIN_ACTION* macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note: Only one of the forms of this macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID* should be used for a particular action, never both.

namespace **hpx**

namespace **actions**

hpx/actions_base/basic_action_fwd.hpp

Defined in header hpx/actions_base/basic_action_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **actions**

template<typename **Component**, typename **Signature**, typename **Derived**>

struct **basic_action**

#include <basic_action_fwd.hpp>

Template Parameters

- **Component** – component type
- **Signature** – return type and arguments
- **Derived** – derived action class

hpx/actions_base/component_action.hpp

Defined in header hpx/actions_base/component_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_DEFINE_COMPONENT_ACTION(...)

Registers a member function of a component as an action type with HPX.

The macro *HPX_DEFINE_COMPONENT_ACTION* can be used to register a member function of a component as an action type named *action_type*.

The parameter *component* is the type of the component exposing the member function *func* which should be associated with the newly defined action type. The parameter `action_type` is the name of the action type to register with HPX.

```
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
```

(continues on next page)

(continued from previous page)

```
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::component_base<server>
    {
        void print_greeting() const
        {
            hpx::cout << "Hey, how are you?\n" << std::flush;
        }

        // Component actions need to be declared, this also defines the
        // type 'print_greeting_action' representing the action.
        HPX_DEFINE_COMPONENT_ACTION(server, print_greeting,
            print_greeting_action);
    };
}
```

Example:

The first argument must provide the type name of the component the action is defined for.

The second argument must provide the member function name the action should wrap.

The default value for the third argument (the typename of the defined action) is derived from the name of the function (as passed as the second argument) by appending '_action'. The third argument can be omitted only if the second argument with an appended suffix '_action' resolves to a valid, unqualified C++ type name.

Note: The macro *HPX_DEFINE_COMPONENT_ACTION* can be used with 2 or 3 arguments. The third argument is optional.

namespace **hpx**

namespace **actions**

hpx/actions_base/lambda_to_action.hpp

Defined in header hpx/actions_base/lambda_to_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/actions_base/plain_action.hpp

Defined in header hpx/actions_base/plain_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_DEFINE_PLAIN_ACTION(...)

Defines a plain action type.

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }

    // This will define the action type 'app::some_global_action' which
    // represents the function 'app::some_global_function'.
    HPX_DEFINE_PLAIN_ACTION(some_global_function, some_global_action);
}
```

Example:

Note: Usually this macro will not be used in user code unless the intent is to avoid defining the action_type in global namespace. Normally, the use of the macro *HPX_PLAIN_ACTION* is recommended.

Note: The macro *HPX_DEFINE_PLAIN_ACTION* can be used with 1 or 2 arguments. The second argument is optional. The default value for the second argument (the typename of the defined action) is derived from the name of the function (as passed as the first argument) by appending '_action'. The second argument can be omitted only if the first argument with an appended suffix '_action' resolves to a valid, unqualified C++ type name.

HPX_DECLARE_PLAIN_ACTION(...)

Declares a plain action type.

HPX_PLAIN_ACTION(...)

Defines a plain action type based on the given function *func* and registers it with HPX.

The macro *HPX_PLAIN_ACTION* can be used to define a plain action (e.g. an action encapsulating a global or free function) based on the given function *func*. It defines the action type *name* representing the given function. This macro additionally registers the newly define action type with HPX.

The parameter `func` is a global or free (non-member) function which should be encapsulated into a plain action. The parameter `name` is the name of the action type defined by this macro.

```
namespace app {
    void some_global_function(double d) {
        cout << d;
    }
}

// This will define the action type 'some_global_action' which
```

(continues on next page)

(continued from previous page)

```
// represents the function 'app::some_global_function'.
HPX_PLAIN_ACTION(app::some_global_function, some_global_action)
```

Example:

Note: The macro *HPX_PLAIN_ACTION* has to be used at global namespace even if the wrapped function is located in some other namespace. The newly defined action type is placed into the global namespace as well.

Note: The macro *HPX_PLAIN_ACTION_ID* can be used with 1, 2, or 3 arguments. The second and third arguments are optional. The default value for the second argument (the typename of the defined action) is derived from the name of the function (as passed as the first argument) by appending '_action'. The second argument can be omitted only if the first argument with an appended suffix '_action' resolves to a valid, unqualified C++ type name. The default value for the third argument is *hpx::components::factory_state::check*.

Note: Only one of the forms of this macro *HPX_PLAIN_ACTION* or *HPX_PLAIN_ACTION_ID* should be used for a particular action, never both.

HPX_PLAIN_ACTION_ID(func, name, id)

Defines a plain action type based on the given function *func* and registers it with HPX.

The macro *HPX_PLAIN_ACTION_ID* can be used to define a plain action (e.g. an action encapsulating a global or free function) based on the given function *func*. It defines the action type *actionname* representing the given function.

The parameter *actionid* specifies a unique integer value which will be used to represent the action during serialization.

The parameter `func` is a global or free (non-member) function which should be encapsulated into a plain action. The parameter `name` is the name of the action type defined by this macro.

The second parameter has to be usable as a plain (non-qualified) C++ identifier, it should not contain special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

```
namespace app {
    void some_global_function(double d) {
        cout << d;
    }
}

// This will define the action type 'some_global_action' which
// represents the function 'app::some_global_function'.
HPX_PLAIN_ACTION_ID(app::some_global_function, some_global_action,
    some_unique_id);
```

Example:

Note: The macro *HPX_PLAIN_ACTION_ID* has to be used at global namespace even if the wrapped function is located in some other namespace. The newly defined action type is placed into the global namespace as well.

Note: Only one of the forms of this macro *HPX_PLAIN_ACTION* or *HPX_PLAIN_ACTION_ID* should be used for a particular action, never both.

namespace **hpx**

namespace **actions**

namespace **traits**

hpx/actions_base/preassigned_action_id.hpp

Defined in header hpx/actions_base/preassigned_action_id.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **actions**

hpx/actions_base/traits/action_remote_result.hpp

Defined in header hpx/actions_base/traits/action_remote_result.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **traits**

Typedefs

template<typename **Result**>

using **action_remote_result_t** = typename *action_remote_result*<*Result*>::type

template<typename **Result**>

struct **action_remote_result** : public detail::action_remote_result_customization_point<*Result*>

agas

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/agas/addressing_service.hpp

Defined in header hpx/agas/addressing_service.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **agas**

struct **addressing_service**

Public Types

using **component_id_type** = *components*::component_type

using **iterate_names_return_type** = *std*::map<*std*::string, *hpx*::id_type>

using **iterate_types_function_type** = *hpx*::*function*<void(*std*::string const&, *components*::component_type), true>

using **mutex_type** = *hpx*::*spinlock*

using **gva_cache_type** = *hpx*::*util*::*cache*::*lru_cache*<gva_cache_key, gva, *hpx*::*util*::*cache*::*statistics*::local_full_statistics>

using **migrated_objects_table_type** = *std*::set<*naming*::gid_type>

using **refcnt_requests_type** = *std*::map<*naming*::gid_type, *std*::int64_t>

using **resolved_localities_type** = *std*::map<*naming*::gid_type, *parcelset*::endpoints_type>

Public Functions

HPX_NON_COPYABLE(*addressing_service*)

explicit **addressing_service**(*util*::runtime_configuration const &ini_)

~addressing_service() = default

void **bootstrap**(*parcelset*::endpoints_type const &endpoints, *util*::runtime_configuration &rtcfg)

void **initialize**(*std*::uint64_t rts_lva)

void **adjust_local_cache_size**(*std*::size_t) const

Adjust the size of the local AGAS Address resolution cache.

inline state **get_status**() const

inline void **set_status**(state new_state)

inline *naming*::gid_type const &**get_local_locality**(*error_code*& = *throws*) const

void **set_local_locality**(*naming*::gid_type const &g)

void **register_console**(*parcelset*::endpoints_type const &eps)

inline bool **is_bootstrap**() const

inline bool **is_console**() const

Returns whether this *addressing_service* represents the console locality.

inline bool **is_connecting**() const

Returns whether this *addressing_service* is connecting to a running application.

bool **resolve_locally_known_addresses**(*naming*::gid_type const &id, *naming*::address &addr) const

void **register_server_instances**()

void **garbage_collect_non_blocking**(*error_code* &ec = *throws*)

void **garbage_collect**(*error_code* &ec = *throws*)

inline *server*::*primary_namespace* &**get_local_primary_namespace_service**()

inline *naming*::address::address_type **get_primary_ns_lva**() const

inline *naming*::address::address_type **get_symbol_ns_lva**() const

inline *server*::component_namespace ***get_local_component_namespace_service**() const

inline *server*::locality_namespace ***get_local_locality_namespace_service**() const

inline *server*::symbol_namespace &**get_local_symbol_namespace_service**() const

inline *naming*::address::address_type **get_runtime_support_lva**() const

std::uint64_t **get_cache_entries**(bool) const

std::uint64_t **get_cache_hits**(bool) const

std::uint64_t **get_cache_misses**(bool) const

std::uint64_t **get_cache_evictions**(bool) const

std::uint64_t **get_cache_insertions**(bool) const

std::uint64_t **get_cache_get_entry_count**(bool reset) const

std::uint64_t **get_cache_insertion_entry_count**(bool reset) const *std*::uint64_t **get_cache_update_entry_count**(bool reset) const

std::uint64_t **get_cache_erase_entry_count**(bool reset) const

std::uint64_t **get_cache_get_entry_time**(bool reset) const

std::uint64_t **get_cache_insertion_entry_time**(bool reset) const

std::uint64_t **get_cache_update_entry_time**(bool reset) const

std::uint64_t **get_cache_erase_entry_time**(bool reset) const

bool **register_locality**(*parcelset*::endpoints_type const &endpoints, *naming*::gid_type &prefix, *std*::uint32_t num_threads, *error_code* &ec = *throws*)

Add a locality to the runtime.

parcelset::endpoints_type const &**resolve_locality**(*naming*::gid_type const &gid, *error_code* &ec = *throws*)

Resolve a locality to its prefix.

Returns Returns an empty vector if the locality is not registered.

bool **has_resolved_locality**(*naming*::gid_type const &gid)

bool **unregister_locality**(*naming*::gid_type const &gid, *error_code* &ec = *throws*)

Remove a locality from the runtime.

void **remove_resolved_locality**(*naming*::gid_type const &gid)

remove given locality from locality cache

bool **get_console_locality**(*naming*::gid_type &locality, *error_code* &ec = *throws*)

Get locality locality_id of the console locality.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **locality** – [out] The locality_id value uniquely identifying the console locality. This is valid only, if the return value of this function is true.
- **try_cache** – [in] If this is set to true the console is first tried to be found in the local cache. Otherwise this function will always query AGAS, even if the console locality_id is already known locally.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if a console locality_id exists and returns *false* otherwise.

bool **get_localities**(*std*::vector<*naming*::gid_type> &locality_ids, *components*::component_type type, *error_code* &ec = *throws*) const

Query for the locality_ids of all known localities.

This function returns the locality_ids of all localities known to the AGAS server or all localities having a registered factory for a given component type.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise, it throws an instance of hpx::exception.

Parameters

- **locality_ids** – [out] The vector will contain the prefixes of all localities registered with the AGAS server. The returned vector holds the prefixes representing the runtime_support components of these localities.
- **type** – [in] The component type will be used to determine the set of prefixes having a registered factory for this component. The default value for this parameter is *components::component_enum_type::invalid*, which will return prefixes of all localities.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

inline bool **get_localities**(*std*::vector<*naming*::gid_type> &locality_ids, *error_code* &ec = *throws*) const

hpx::*future*<*std*::uint32_t> **get_num_localities_async**(*components*::component_type type = to_int(*hpx*::*components*::*component_enum_type*::*invalid*)) const

Query for the number of all known localities.

This function returns the number of localities known to the AGAS server or the number of localities having a registered factory for a given component type.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise, it throws an instance of hpx::exception.

Parameters

- **type** – [in] The component type will be used to determine the set of prefixes having a registered factory for this component. The default value for this parameter is *components::component_type::invalid*, which will return prefixes of all localities.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

std::uint32_t **get_num_localities**(*components*::component_type type, *error_code* &ec = *throws*) const

inline *std*::uint32_t **get_num_localities**(*error_code* &ec = *throws*) const

hpx::*future*<*std*::uint32_t> **get_num_overall_threads_async**() const

std::uint32_t **get_num_overall_threads**(*error_code* &ec = *throws*) const

hpx::*future*<*std*::vector<*std*::uint32_t>> **get_num_threads_async**() const

std::vector<*std*::uint32_t> **get_num_threads**(*error_code* &ec = *throws*) const

components::component_type **get_component_id**(*std*::string const &name, *error_code* &ec = *throws*) const

Return a unique id usable as a component type.

This function returns the component type id associated with the given component name. If this is the first request for this component name a new unique id will be created.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **name** – [in] The component name (string) to get the component type for.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The function returns the currently associated component type. Any error results in an exception thrown from this function.

void **iterate_types**(*iterate_types_function_type* const &f, *error_code* &ec = *throws*) const

std::string **get_component_type_name**(*components*::component_type id, *error_code* &ec = *throws*) const

inline *components*::component_type **register_factory**(*naming*::gid_type const &locality_id, *std*::string const &name, *error_code* &ec = *throws*) const

Register a factory for a specific component type.

This function allows to register a component factory for a given locality and component type.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **locality_id** – [in] The locality value uniquely identifying the given locality the factory needs to be registered for.
- **name** – [in] The component name (string) to register a factory for the given component type for.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The function returns the currently associated component type. Any error results in an exception thrown from this function. The returned component type is the same as if the function *get_component_id* was called using the same component name.

components::component_type **register_factory**(*std*::uint32_t locality_id, *std*::string const &name, *error_code* &ec = *throws*) const

bool **get_id_range**(*std*::uint64_t count, *naming*::gid_type &lower_bound, *naming*::gid_type &upper_bound, *error_code* &ec = *throws*)

Get unique range of freely assignable global ids.

Every locality needs to be able to assign global ids to different components without having to consult the AGAS server for every id to generate. This function can be called to preallocate a range of ids usable for this purpose.

Note: This function assigns a range of global ids usable by the given locality for newly created components. Any of the returned global ids still has to be bound to a local address, either by calling *bind* or *bind_range*.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **l** – [in] The locality the locality id needs to be generated for. Repeating calls using the same locality results in identical locality_id values.

- **count** – [in] The number of global ids to be generated.
- **lower_bound** – [out] The lower bound of the assigned id range. The returned value can be used as the first id to assign. This is valid only, if the return value of this function is true.
- **upper_bound** – [out] The upper bound of the assigned id range. The returned value can be used as the last id to assign. This is valid only, if the return value of this function is true.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if a new range has been generated (it has been called for the first time for the given locality) and returns *false* if this locality already got a range assigned in an earlier call. Any error results in an exception thrown from this function.

inline bool **bind_local**(*naming*::gid_type const &id, *naming*::address const &addr, *error_code* &ec = *throws*)

Bind a global address to a local address.

Every element in the HPX namespace has a unique global address (global id). This global address has to be associated with a concrete local address to be able to address an instance of a component using its global address.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: Binding a gid to a local address sets its global reference count to one.

Parameters

- **id** – [in] The global address which has to be bound to the local address.
- **addr** – [in] The local address to be bound to the global address.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true*, if this global id got associated with an local address. It returns *false* otherwise.

inline *hpx*::*future*<bool> **bind_async**(*naming*::gid_type const &id, *naming*::address const &addr, *std*::uint32_t locality_id)

inline *hpx*::*future*<bool> **bind_async**(*naming*::gid_type const &id, *naming*::address const &addr, *naming*::gid_type const &locality)

bool **bind_range_local**(*naming*::gid_type const &lower_id, *std*::uint64_t count, *naming*::address const &baseaddr, *std*::uint64_t offset, *error_code* &ec = *throws*)

Bind unique range of global ids to given base address.

Every locality needs to be able to bind global ids to different components without having to consult the AGAS server for every id to bind. This function can be called to bind a range of consecutive global ids to a range of consecutive local addresses (separated by a given *offset*).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: Binding a gid to a local address sets its global reference count to one.

Parameters
- **lower_id** – [in] The lower bound of the assigned id range. The value can be used as the first id to assign.
- **count** – [in] The number of consecutive global ids to bind starting at *lower_id*.
- **baseaddr** – [in] The local address to bind to the global id given by *lower_id*. This is the base address for all additional local addresses to bind to the remaining global ids.
- **offset** – [in] The offset to use to calculate the local addresses to be bound to the range of global ids.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true*, if the given range was successfully bound. It returns *false* otherwise.

hpx::*future*<bool> **bind_range_async**(*naming*::gid_type const &lower_id, *std*::uint64_t count, *naming*::address const &baseaddr, *std*::uint64_t offset, *naming*::gid_type const &locality)

inline *hpx*::*future*<bool> **bind_range_async**(*naming*::gid_type const &lower_id, *std*::uint64_t count, *naming*::address const &baseaddr, *std*::uint64_t offset, *std*::uint32_t locality_id)

inline bool **unbind_local**(*naming*::gid_type const &id, *error_code* &ec = *throws*)

Unbind a global address.

Remove the association of the given global address with any local address, which was bound to this global address. Additionally it returns the local address which was bound at the time of this call.

Note: You can unbind only global ids bound using the function *bind*. Do not use this function to unbind any of the global ids bound using *bind_range*.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will raise an error if the global reference count of the given gid is not zero!

Parameters
- **id** – [in] The global address (id) for which the association has to be removed.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The function returns *true* if the association has been removed, and it returns *false* if no association existed. Any error results in an exception thrown from this function.

inline bool **unbind_local**(*naming*::gid_type const &id, *naming*::address &addr, *error_code* &ec = *throws*)

Unbind a global address.

Remove the association of the given global address with any local address, which was bound to this global address. Additionally it returns the local address which was bound at the time of this call.

Note: You can unbind only global ids bound using the function *bind*. Do not use this function to unbind any of the global ids bound using *bind_range*.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will raise an error if the global reference count of the given gid is not zero!

Parameters

- **id** – [in] The global address (id) for which the association has to be removed.
- **addr** – [out] The local address which was associated with the given global address (id). This is valid only if the return value of this function is true.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The function returns *true* if the association has been removed, and it returns *false* if no association existed. Any error results in an exception thrown from this function.

inline bool **unbind_range_local**(*naming*::gid_type const &lower_id, *std*::uint64_t count, *error_code* &ec = *throws*)

Unbind the given range of global ids.

Note: You can unbind only global ids bound using the function *bind_range*. Do not use this function to unbind any of the global ids bound using *bind*.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will raise an error if the global reference count of the given gid is not zero!

Parameters

- **lower_id** – [in] The lower bound of the assigned id range. The value must the first id of the range as specified to the corresponding call to *bind_range*.
- **count** – [in] The number of consecutive global ids to unbind starting at *lower_id*. This number must be identical to the number of global ids bound by the corresponding call to *bind_range*
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if a new range has been generated (it has been called for the first time for the given locality) and returns *false* if this locality already got a range assigned in an earlier call. Any error results in an exception thrown from this function.

bool **unbind_range_local**(*naming*::gid_type const &lower_id, *std*::uint64_t count, *naming*::address &addr, *error_code* &ec = *throws*)

Unbind the given range of global ids.

Note: You can unbind only global ids bound using the function *bind_range*. Do not use this function to unbind any of the global ids bound using *bind*.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will raise an error if the global reference count of the given gid is not zero!

Parameters

- **lower_id** – [in] The lower bound of the assigned id range. The value must the first id of the range as specified to the corresponding call to *bind_range*.
- **count** – [in] The number of consecutive global ids to unbind starting at *lower_id*. This number must be identical to the number of global ids bound by the corresponding call to *bind_range*
- **addr** – [out] The local address which was associated with the given global address (id). This is valid only if the return value of this function is true.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if a new range has been generated (it has been called for the first time for the given locality) and returns *false* if this locality already got a range assigned in an earlier call.

hpx::*future*<*naming*::address> **unbind_range_async**(*naming*::gid_type const &lower_id, *std*::uint64_t count = 1)

inline bool **is_local_address_cached**(*naming*::gid_type const &id, *error_code* &ec = *throws*)

Test whether the given address refers to a local object.

This function will test whether the given address refers to an object living on the locality of the caller.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The address to test.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if the passed address refers to an object which lives on the locality of the caller.

bool **is_local_address_cached**(*naming*::gid_type const &id, *naming*::address &addr, *error_code* &ec = *throws*) bool **is_local_address_cached**(*naming*::gid_type const &id, *naming*::address &addr, *std*::pair<bool, *components*::pinned_ptr> &r, *hpx*::*move_only_function*<*std*::pair<bool, *components*::pinned_ptr>(*naming*::address const&)> &&f, *error_code* &ec = *throws*)

bool **is_local_lva_encoded_address**(*std*::uint64_t msb) const

inline bool **resolve_local**(*naming*::gid_type const &id, *naming*::address &addr, *error_code* &ec = *throws*)

Resolve a given global address (*id*) to its associated local address.

This function returns the local address which is currently associated with the given global address (*id*).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The global address (*id*) for which the associated local address should be returned.
- **addr** – [out] The local address which currently is associated with the given global address (id), this is valid only if the return value of this function is true.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns *true* if the global address has been resolved successfully (there exists an association to a local address) and the associated local address has been returned. The function returns *false* if no association exists for the given global address. Any error results in an exception thrown from this function.

inline bool **resolve_local**(*hpx*::id_type const &id, *naming*::address &addr, *error_code* &ec = *throws*)

inline *naming*::address **resolve_local**(*naming*::gid_type const &id, *error_code* &ec = *throws*)

inline *naming*::address **resolve_local**(*hpx*::id_type const &id, *error_code* &ec = *throws*)

hpx::future_or_value<*naming*::address> **resolve_async**(*naming*::gid_type const &id)

inline *hpx*::future_or_value<*naming*::address> **resolve_async**(*hpx*::id_type const &id)

hpx::future_or_value<id_type> **get_colocation_id_async**(*hpx*::id_type const &id)

bool **resolve_full_local**(*naming*::gid_type const &id, *naming*::address &addr, *error_code* &ec = *throws*)

inline bool **resolve_full_local**(*hpx*::id_type const &id, *naming*::address &addr, *error_code* &ec = *throws*)

inline *naming*::address **resolve_full_local**(*naming*::gid_type const &id, *error_code* &ec = *throws*)

inline *naming*::address **resolve_full_local**(*hpx*::id_type const &id, *error_code* &ec = *throws*)

hpx::future_or_value<*naming*::address> **resolve_full_async**(*naming*::gid_type const &id)

inline *hpx*::future_or_value<*naming*::address> **resolve_full_async**(*hpx*::id_type const &id)

bool **resolve_cached**(*naming*::gid_type const &id, *naming*::address &addr, *error_code* &ec = *throws*)

inline bool **resolve_cached**(*hpx*::id_type const &id, *naming*::address &addr, *error_code* &ec = *throws*)

inline bool **resolve_local**(*naming*::gid_type const *gids, *naming*::address *addrs, *std*::size_t size, *hpx*::detail::dynamic_bitset<> &locals, *error_code* &ec = *throws*)

bool **resolve_full_local**(*naming*::gid_type const *gids, *naming*::address *addrs, *std*::size_t size, *hpx*::detail::dynamic_bitset<> &locals, *error_code* &ec = *throws*)

bool **resolve_cached**(*naming*::gid_type const *gids, *naming*::address *addrs, *std*::size_t size, *hpx*::detail::dynamic_bitset<> &locals, *error_code* &ec = *throws*)

hpx::future_or_value<*std*::int64_t> **incref_async**(*naming*::gid_type const &gid, *std*::int64_t credits = 1, *hpx*::id_type const &keep_alive = *hpx*::invalid_id)

Increment the global reference count for the given id.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **gid** – [in] The global address (id) for which the global reference count has to be incremented.
- **credits** – [in] The number of reference counts to add for the given id.
- **keep_alive** – [in] Id to keep alive (if valid)
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns Whether the operation was successful.

inline *std*::int64_t **incref**(*naming*::gid_type const &gid, *std*::int64_t credits = 1, *error_code* &ec = *throws*)

void **decref**(*naming*::gid_type const &id, *std*::int64_t credits = 1, *error_code* &ec = *throws*)

Decrement the global reference count for the given id.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **id** – [in] The global address (id) for which the global reference count has to be decremented.
- **t** – [out] If this was the last outstanding global reference for the given gid (the return value of this function is zero), t will be set to the component type of the corresponding element. Otherwise t will not be modified.
- **credits** – [in] The number of reference counts to add for the given id.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global reference count after the decrement.

hpx::future<iterate_names_return_type> **iterate_ids**(std::string const &pattern) const

Invoke the supplied *hpx::function* for every registered global name.

This function iterates over all registered global ids and returns every found entry matching the given name pattern. Any error results in an exception thrown (or reported) from this function.

Parameters **pattern** – [in] pattern (possibly using wildcards) to match all existing entries against

bool **register_name**(std::string const &name, naming::gid_type const &id, error_code &ec = throws) const

Register a global name with a global address (id)

This function registers an association between a global name (string) and a global address (id) usable with one of the functions above (bind, unbind, and resolve).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **name** – [in] The global name (string) to be associated with the global address.
- **id** – [in] The global address (id) to be associated with the global address.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The function returns *true* if the global name was registered. It returns false if the global name is not registered.

hpx::future<bool> **register_name_async**(std::string const &name, hpx::id_type const &id) const

bool **register_name**(std::string const &name, hpx::id_type const &id, error_code &ec = throws) const

hpx::future<hpx::id_type> **unregister_name_async**(std::string const &name) const

Unregister a global name (release any existing association)

This function releases any existing association of the given global name with a global address (id).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **name** – [in] The global name (string) for which any association with a global address (id) has to be released.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The function returns *true* if an association of this global name has been released, and it returns *false*, if no association existed. Any error results in an exception thrown from this function.

hpx::id_type **unregister_name**(std::string const &name, error_code &ec = throws) const

hpx::future<hpx::id_type> **resolve_name_async**(std::string const &name) const

Query for the global address associated with a given global name.

This function returns the global address associated with the given global name.

This function returns true if it returned global address (id), which is currently associated with the given global name, and it returns false, if currently there is no association for this global name. Any error results in an exception thrown from this function.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **name** – [in] The global name (string) for which the currently associated global address has to be retrieved.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns [out] The id currently associated with the given global name (valid only if the return value is true).

hpx::id_type **resolve_name**(*std*::string const &name, *error_code* &ec = *throws*) const

future<*hpx*::id_type> **on_symbol_namespace_event**(*std*::string const &name, bool call_for_past_events = false) const

Install a listener for a given symbol namespace event.

This function installs a listener for a given symbol namespace event. It returns a future which becomes ready as a result of the listener being triggered.

Note: The only event type which is currently supported is *symbol_ns_bind*, i.e. the listener is triggered whenever a global id is registered with the given name.

Parameters
- **name** – [in] The global name (string) for which the given event should be triggered.
- **evt** – [in] The event for which a listener should be installed.
- **call_for_past_events** – [in, optional] Trigger the listener even if the given event has already happened in the past. The default for this parameter is *false*.

Returns A future instance encapsulating the global id which is causing the registered listener to be triggered.

void **update_cache_entry**(*naming*::gid_type const &gid, *gva* const &gva, *error_code* &ec = *throws*)

> **Warning:** This function is for internal use only. It is dangerous and may break your code if you use it.

inline void **update_cache_entry**(*naming*::gid_type const &gid, *naming*::address const &addr, *std*::uint64_t count = 0, *std*::uint64_t offset = 0, *error_code* &ec = *throws*)

> **Warning:** This function is for internal use only. It is dangerous and may break your code if you use it.

bool **get_cache_entry**(*naming*::gid_type const &gid, *gva* &gva, *naming*::gid_type &idbase, *error_code* &ec = *throws*) const

> **Warning:** This function is for internal use only. It is dangerous and may break your code if you use it.

void **remove_cache_entry**(*naming*::gid_type const &id, *error_code* &ec = *throws*) const

> **Warning:** This function is for internal use only. It is dangerous and may break your code if you use it.

void **clear_cache**(*error_code* &ec = *throws*) const

> **Warning:** This function is for internal use only. It is dangerous and may break your code if you use it.

void **start_shutdown**(*error_code* &ec = *throws*)

hpx::*future*<*std*::pair<*hpx*::id_type, *naming*::address>> **begin_migration**(*hpx*::id_type const &id)

start/stop migration of an object

Returns Current locality and address of the object to migrate

bool **end_migration**(*hpx*::id_type const &id)

std::pair<bool, *components*::pinned_ptr> **was_object_migrated**(*naming*::gid_type const &gid, *hpx*::*move_only_function*<*components*::pinned_ptr(&&f)

Maintain list of migrated objects.

hpx::*future*<void> **mark_as_migrated**(*naming*::gid_type const &gid, *hpx*::*move_only_function*<*std*::pair<bool, *hpx*::*future*<void>>()> &&f, bool expect_to_be_marked_as_migrating)

Mark the given object as being migrated (if the object is unpinned). Delay migration until the object is unpinned otherwise.

void **unmark_as_migrated**(*naming*::gid_type const &gid_, *hpx*::*move_only_function*<void()> &&f)

Remove the given object from the table of migrated objects.

void **pre_cache_endpoints**(*std*::vector<*parcelset*::endpoints_type> const&)

Public Members

mutable hpx::shared_mutex **gva_cache_mtx_**

std::shared_ptr<gva_cache_type> **gva_cache_**

mutable mutex_type **migrated_objects_mtx_**

migrated_objects_table_type **migrated_objects_table_**

mutable mutex_type **console_cache_mtx_**

std::uint32_t **console_cache_**

const std::size_t **max_refcnt_requests_**

mutex_type **refcnt_requests_mtx_**

std::size_t **refcnt_requests_count_**

bool **enable_refcnt_caching_**

std::shared_ptr<refcnt_requests_type> **refcnt_requests_**

const service_mode **service_type**

const runtime_mode **runtime_type**

const bool **caching_**

const bool **range_caching_**

const threads::thread_priority **action_priority_**

std::uint64_t **rts_lva_**

std::unique_ptr<component_namespace> **component_ns_**

std::unique_ptr<locality_namespace> **locality_ns_**

symbol_namespace **symbol_ns_**

primary_namespace **primary_ns_**

std::atomic<*hpx*::state> **state_**

naming::gid_type **locality_**

mutable *hpx*::*shared_mutex* **resolved_localities_mtx_**

resolved_localities_type **resolved_localities_**

Public Static Functions

static *std*::int64_t **synchronize_with_async_incref**(*std*::int64_t old_credit, *hpx*::id_type const &id, *std*::int64_t compensated_credit)

Protected Functions

void **launch_bootstrap**(*parcelset*::endpoints_type const &endpoints, *util*::runtime_configuration &rtcfg)

naming::address **resolve_full_postproc**(*naming*::gid_type const &id, primary_namespace::resolved_type const&)

bool **bind_postproc**(*naming*::gid_type const &id, gva const &g, *future*<bool> f)

bool **was_object_migrated_locked**(*naming*::gid_type const &id)

Maintain list of migrated objects.

Private Functions

void **send_refcnt_requests**(*std*::unique_lock<*mutex_type*> &l, *error_code* &ec = *throws*)

Assumes that *refcnt_requests_mtx_* is locked.

void **send_refcnt_requests_non_blocking**(*std*::unique_lock<*mutex_type*> &l, *error_code* &ec)

Assumes that *refcnt_requests_mtx_* is locked.

std::vector<*hpx*::*future*<*std*::vector<*std*::int64_t>>> **send_refcnt_requests_async**(*std*::unique_lock<*mutex_type*> &l)

Assumes that *refcnt_requests_mtx_* is locked.

void **send_refcnt_requests_sync**(*std*::unique_lock<*mutex_type*> &l, *error_code* &ec)

Assumes that *refcnt_requests_mtx_* is locked.

agas_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/agas_base/server/primary_namespace.hpp

Defined in header hpx/agas_base/server/primary_namespace.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Variables

**HPX_ACTION_USES_MEDIUM_STACK(hpx::agas::server::primary_namespace::allocate_action) HPX_REGISTER_ACTION_
hpx::naming::address > std_pair_address_id_type**

namespace **hpx**

namespace **agas**

Functions

naming::gid_type **bootstrap_primary_namespace_gid**()

hpx::id_type **bootstrap_primary_namespace_id**()

namespace **server**

AGAS's primary namespace maps 128-bit global identifiers (GIDs) to resolved addresses.

The following is the canonical description of the partitioning of AGAS's primary namespace.

```
|-----MSB------||------LSB-----|
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
|prefix||RC||----identifier----|

MSB        - Most significant bits (bit 64 to bit 127)
LSB        - Least significant bits (bit 0 to bit 63)
prefix     - Highest 32 bits (bit 96 to bit 127) of the MSB. Each
             locality is assigned a prefix. This creates a 96-bit
             address space for each locality.
RC         - Bit 88 to bit 92 of the MSB. This is the log2 of the number
             of reference counting credits on the GID.
             Bit 93 is used by the locking scheme for gid_types.
             Bit 94 is a flag which is set if the credit value is valid.
             Bit 95 is a flag that is set if a GID's credit count is
             ever split (e.g. if the GID is ever passed to another
             locality).
           - Bit 87 marks the gid such that it will not be stored in
             any of the AGAS caches. This is used mainly for ids
```

(continues on next page)

(continued from previous page)

```
                which represent 'one-shot' objects (like promises).
identifier - Bit 64 to bit 86 of the MSB, and the entire LSB. The
                content of these bits depends on the component type of
                the underlying object. For all user-defined components,
                these bits contain a unique 88-bit number which is
                assigned sequentially for each locality. For
                \a hpx#components#component_enum_type#runtime_support
                the high 24 bits are zeroed and the low 64 bits hold the
                LVA of the component.
```

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

```
00000000xxxxxxxxxxxxxxxxxxxxxxxx
    Historically unused address space reserved for future use.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxx
    Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxx
    Prefix of the bootstrap AGAS locality.
00000001000000010000000000000001
    Address of the primary_namespace component on the bootstrap AGAS
    locality.
00000001000000010000000000000002
    Address of the component_namespace component on the bootstrap AGAS
    locality.
00000001000000010000000000000003
    Address of the symbol_namespace component on the bootstrap AGAS
    locality.
00000001000000010000000000000004
    Address of the locality_namespace component on the bootstrap AGAS
    locality.
```

Note: The layout of the address space is implementation defined, and subject to change. Never write application code that relies on the internal layout of GIDs. AGAS only guarantees that all assigned GIDs will be unique.

Variables

static constexpr char const *const **primary_namespace_service_name** = "primary/"

struct **primary_namespace** : public *components*::*fixed_component_base*<*primary_namespace*>

Public Types

using **mutex_type** = *hpx*::*spinlock*

using **base_type** = *components*::*fixed_component_base*<*primary_namespace*>

using **component_type** = *std*::int32_t

using **gva_table_data_type** = *std*::pair<gva, *naming*::gid_type>

using **gva_table_type** = *std*::map<*naming*::gid_type, *gva_table_data_type*>

using **refcnt_table_type** = *std*::map<*naming*::gid_type, *std*::int64_t>

using **resolved_type** = *hpx*::*tuple*<*naming*::gid_type, gva, *naming*::gid_type>

Public Functions

inline *mutex_type* &**mutex**()

void **wait_for_migration_locked**(*std*::unique_lock<*mutex_type*> &l, *naming*::gid_type const &id, *error_code* &ec)

inline **primary_namespace**()

void **finalize**() const

inline void **set_local_locality**(*naming*::gid_type const &g)

void **register_server_instance**(char const *servicename, *std*::uint32_t locality_id = *naming*::invalid_locality_id, *error_code* &ec = *throws*)

void **unregister_server_instance**(*error_code* &ec = *throws*) const

bool **bind_gid**(gva const &g, *naming*::gid_type id, *naming*::gid_type const &locality)

std::pair<*hpx*::id_type, *naming*::address> **begin_migration**(*naming*::gid_type id)

bool **end_migration**(*naming*::gid_type const &id)

resolved_type **resolve_gid**(*naming*::gid_type const &id)

hpx::id_type **colocate**(*naming*::gid_type const &id)

naming::address **unbind_gid**(*std*::uint64_t count, *naming*::gid_type id)

std::int64_t **increment_credit**(*std*::int64_t credits, *naming*::gid_type lower, *naming*::gid_type upper)

std::vector<*std*::int64_t> **decrement_credit**(*std*::vector<*hpx*::*tuple*<*std*::int64_t, *naming*::gid_type, *naming*::gid_type>> const &requests)

std::pair<*naming*::gid_type, *naming*::gid_type> **allocate**(*std*::uint64_t count)

resolved_type **resolve_gid_locked**(*std*::unique_lock<*mutex_type*> &l, *naming*::gid_type const &gid, *error_code* &ec)

Public Members

counter_data **counter_data_**

Private Types

using **migration_table_type** = *std*::map<*naming*::gid_type, *hpx*::*tuple*<bool, *std*::size_t, *lcos*::*local*::detail::condition_variable>>

using **free_entry_allocator_type** = *util*::internal_allocator<*free_entry*>

using **free_entry_list_type** = *std*::list<*free_entry*, *free_entry_allocator_type*>

Private Functions

resolved_type **resolve_gid_locked_non_local**(*std*::unique_lock<*mutex_type*> &l, *naming*::gid_type const &gid, *error_code* &ec)

void **increment**(*naming*::gid_type const &lower, *naming*::gid_type const &upper, *std*::int64_t const &credits, *error_code* &ec)

void **resolve_free_list**(*std*::unique_lock<*mutex_type*> &l, *std*::list<*refcnt_table_type*::iterator> const &free_list, *free_entry_list_type* &free_entry_list, *naming*::gid_type const &lower, *naming*::gid_type const &upper, *error_code* &ec)

void **decrement_sweep**(*free_entry_list_type* &free_list, *naming*::gid_type const &lower, *naming*::gid_type const &upper, *std*::int64_t credits, *error_code* &ec)

void **free_components_sync**(*free_entry_list_type* const &free_list, *naming*::gid_type const &lower, *naming*::gid_type const &upper, *error_code* &ec) const

Private Members

mutex_type **mutex_**

gva_table_type **gvas_**

refcnt_table_type **refcnts_**

std::string **instance_name_**

naming::gid_type **next_id_**

naming::gid_type **locality_**

migration_table_type **migrating_objects_**

struct **counter_data**

Public Functions

HPX_NON_COPYABLE(*counter_data*)

counter_data() = default

std::int64_t **get_bind_gid_count**(bool)

std::int64_t **get_resolve_gid_count**(bool)

std::int64_t **get_unbind_gid_count**(bool)

std::int64_t **get_increment_credit_count**(bool)

std::int64_t **get_decrement_credit_count**(bool)

std::int64_t **get_allocate_count**(bool)

std::int64_t **get_begin_migration_count**(bool)

std::int64_t **get_end_migration_count**(bool)

std::int64_t **get_overall_count**(bool)

std::int64_t **get_bind_gid_time**(bool)

std::int64_t **get_resolve_gid_time**(bool)

std::int64_t **get_unbind_gid_time**(bool)

std::int64_t **get_increment_credit_time**(bool)

std::int64_t **get_decrement_credit_time**(bool)

std::int64_t **get_allocate_time**(bool)

std::int64_t **get_begin_migration_time**(bool)

std::int64_t **get_end_migration_time**(bool)

std::int64_t **get_overall_time**(bool)

void **increment_bind_gid_count**()

void **increment_resolve_gid_count**()

void **increment_unbind_gid_count**()

void **increment_increment_credit_count**()

void **increment_decrement_credit_count**()

void **increment_allocate_count**()

void **increment_begin_migration_count**()

void **increment_end_migration_count**()

void **enable_all**()

Public Members

api_counter_data **bind_gid_**

api_counter_data **resolve_gid_**

api_counter_data **unbind_gid_**

api_counter_data **increment_credit_**

api_counter_data **decrement_credit_**

api_counter_data **allocate_**

api_counter_data **begin_migration_**

api_counter_data **end_migration_**

struct **api_counter_data**

Public Functions

inline **api_counter_data**()

Public Members

std::atomic<*std*::int64_t> **count_**

std::atomic<*std*::int64_t> **time_**

bool **enabled_**

struct **free_entry**

Public Functions

inline **free_entry**(*agas*::gva const &gva, *naming*::gid_type const &gid, *naming*::gid_type const &loc)

Public Members

agas::gva **gva_**

naming::gid_type **gid_**

naming::gid_type **locality_**

async_colocated

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::get_colocation_id

Defined in header hpx/runtime.hpp[777].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

hpx::id_type **get_colocation_id**(*launch*::sync_policy, *hpx*::id_type const &id, *error_code* &ec = *throws*)

Return the id of the locality where the object referenced by the given id is currently located on.

The function *hpx::get_colocation_id()* returns the id of the locality where the given object is currently located.

See also:

hpx::get_colocation_id()

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **id** – [in] The id of the object to locate.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

[777] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

hpx::*future*<*hpx*::id_type> **get_colocation_id**(*hpx*::id_type const &id)

Asynchronously return the id of the locality where the object referenced by the given id is currently located on.

See also:

hpx::get_colocation_id(launch::sync_policy)

Parameters **id** – [in] The id of the object to locate.

async_distributed

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/async_distributed/base_lco.hpp

Defined in header hpx/async_distributed/base_lco.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<>

struct *hpx*::**get_lva**<*hpx*::*lcos*::*base_lco*>

Public Static Functions

static inline constexpr *hpx*::*lcos*::*base_lco* ***call**(*hpx*::*naming*::address_type lva) noexcept

template<>

struct *hpx*::**get_lva**<*hpx*::*lcos*::*base_lco* const>

Public Static Functions

static inline constexpr *hpx*::*lcos*::*base_lco* const ***call**(*hpx*::*naming*::address_type lva) noexcept

namespace **hpx**

template<> base_lco >

Public Static Functions

static inline constexpr *hpx*::*lcos*::*base_lco* ***call**(*hpx*::*naming*::address_type lva) noexcept

template<> base_lco const >

Public Static Functions

static inline constexpr *hpx*::*lcos*::*base_lco* const ***call**(*hpx*::*naming*::address_type lva) noexcept

namespace **lcos**

class **base_lco**

#include <base_lco.hpp> The base_lco class is the common base class for all LCO's implementing a simple set_event action

Subclassed by *hpx::lcos::base_lco_with_value< Result, RemoteResult, ComponentTag >*, *hpx::lcos::base_lco_with_value< void, void, ComponentTag >*

Public Types

typedef *components*::*managed_component*<*base_lco*> **wrapping_type**

typedef *base_lco* **base_type_holder**

Public Functions

virtual void **set_event**() = 0

virtual void **set_exception**(*std*::exception_ptr const &e)

virtual void **connect**(*hpx*::id_type const&)

virtual void **disconnect**(*hpx*::id_type const&)

virtual **~base_lco**()

Destructor, needs to be virtual to allow for clean destruction of derived objects

void **set_event_nonvirt**()

The *function* set_event_nonvirt is called whenever a *set_event_action* is applied on a instance of a LCO. This function just forwards to the virtual function *set_event*, which is overloaded by the derived concrete LCO.

void **set_exception_nonvirt**(*std*::exception_ptr const &e)

The *function* set_exception is called whenever a *set_exception_action* is applied on a instance of a LCO. This function just forwards to the virtual function *set_exception*, which is overloaded by the derived concrete LCO.

Parameters **e** – [in] The exception encapsulating the error to report to this LCO instance.

void **connect_nonvirt**(*hpx*::id_type const &id)

The *function* connect_nonvirt is called whenever a *connect_action* is applied on a instance of a LCO. This function just forwards to the virtual function *connect*, which is overloaded by the derived concrete LCO.

Parameters **id** – [in] target id

void **disconnect_nonvirt**(*hpx*::id_type const &id)

The *function* disconnect_nonvirt is called whenever a *disconnect_action* is applied on a instance of a LCO. This function just forwards to the virtual function *disconnect*, which is overloaded by the derived concrete LCO.

Parameters **id** – [in] target id

HPX_DEFINE_COMPONENT_DIRECT_ACTION (base_lco, set_event_nonvirt, set_event_action) HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco

Each of the exposed functions needs to be encapsulated into an action type, allowing to generate all required boilerplate code for threads, serialization, etc.

The *set_event_action* may be used to unconditionally trigger any LCO instances, it carries no additional parameters. The *set_exception_action* may be used to transfer arbitrary error information from the remote site to the LCO instance specified as a continuation. This action carries 2 parameters:

Parameters **std::exception_ptr** – [in] The exception encapsulating the error to report to this LCO instance.

set_exception_action HPX_DEFINE_COMPONENT_DIRECT_ACTION (base_lco, connect_nonvirt, connect_action) HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco

The *connect_action* may be used to.

The *set_exception_action* may be used to

Public Members

set_exception_nonvirt

set_exception_action **disconnect_nonvirt**

Public Static Functions

static *components*::component_type **get_component_type**() noexcept

static void **set_component_type**(*components*::component_type type)

hpx/async_distributed/base_lco_with_value.hpp

Defined in header hpx/async_distributed/base_lco_with_value.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION(...)

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_(...)

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION2(Value, RemoteValue, Name)

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_1(Value)

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_2(Value, Name)

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_3(Value, RemoteValue, Name)

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_4(Value, RemoteValue, Name, Tag)

HPX_REGISTER_BASE_LCO_WITH_VALUE(...)

HPX_REGISTER_BASE_LCO_WITH_VALUE_(...)

HPX_REGISTER_BASE_LCO_WITH_VALUE_1(Value)

HPX_REGISTER_BASE_LCO_WITH_VALUE_2(Value, Name)

HPX_REGISTER_BASE_LCO_WITH_VALUE_3(Value, RemoteValue, Name)

HPX_REGISTER_BASE_LCO_WITH_VALUE_4(Value, RemoteValue, Name, Tag)

HPX_REGISTER_BASE_LCO_WITH_VALUE_ID(...)

HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_(...)

HPX_REGISTER_BASE_LCO_WITH_VALUE_ID2(Value, RemoteValue, Name, ActionIdGet, ActionIdSet)

HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_4(Value, Name, ActionIdGet, ActionIdSet)

HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_5(Value, RemoteValue, Name, ActionIdGet, ActionIdSet)

HPX_REGISTER_BASE_LCO_WITH_VALUE_ID_6(Value, RemoteValue, Name, ActionIdGet, ActionIdSet, Tag)

namespace **hpx**

namespace **components**

namespace **lcos**

template<typename **Result**, typename **RemoteResult**, typename **ComponentTag**>

class **base_lco_with_value** : public *hpx*::*lcos*::*base_lco*, public *ComponentTag*

#include <base_lco_with_value.hpp> The base_lco_with_value class is the common base class for all LCO's synchronizing on a value. The *RemoteResult* template argument should be set to the type of the argument expected for the set_value action.

Template Parameters

- **RemoteResult** – The type of the result value to be carried back to the LCO instance.
- **ComponentTag** – The tag type representing the type of the component (either component_tag or managed_component_tag).

Public Types

using **wrapping_type** = typename detail::base_lco_wrapping_type<*ComponentTag*, *base_lco_with_value*>::type

using **base_type_holder** = *base_lco_with_value*

Public Functions

inline void **set_value_nonvirt**(*RemoteResult* &&result)

The *function* set_value_nonvirt is called whenever a *set_value_action* is applied on this LCO instance. This function just forwards to the virtual function *set_value*, which is overloaded by the derived concrete LCO.

Parameters **result** – [in] The result value to be transferred from the remote operation back to this LCO instance.

inline *Result* **get_value_nonvirt**()

The *function* get_result_nonvirt is called whenever a *get_result_action* is applied on this LCO instance. This function just forwards to the virtual function *get_result*, which is overloaded by the derived concrete LCO.

HPX_DEFINE_COMPONENT_DIRECT_ACTION (base_lco_with_value, set_value_nonvirt, set_value_action) HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco_with_value

The *set_value_action* may be used to trigger any LCO instances while carrying an additional parameter of any type.

RemoteResult is taken by rvalue ref. This allows for perfect forwarding. When the action thread function is created, the values are moved into the called function. If we took it by const lvalue reference, we would disable the possibility to further move the result to the designated destination.

Parameters **RemoteResult** – [in] The type of the result to be transferred back to this LCO instance. The *get_value_action* may be used to query the value this LCO instance exposes as its 'result' value.

Public Members

get_value_nonvirt

Public Static Functions

static inline *components*::component_type **get_component_type**() noexcept

static inline void **set_component_type**(*components*::component_type type)

Protected Types

using **result_type** = *std*::conditional_t<*std*::is_void_v<*Result*>, *util*::unused_type, *Result*>

Protected Functions

~base_lco_with_value() override = default
  Destructor, needs to be virtual to allow for clean destruction of derived objects

inline virtual void **set_event**() override

virtual void **set_value**(*RemoteResult* &&result) = 0

virtual *result_type* **get_value**() = 0

inline virtual *result_type* **get_value**(*error_code*&)

template<typename **ComponentTag**>

class **base_lco_with_value**<void, void, *ComponentTag*> : public *hpx*::*lcos*::*base_lco*, public *ComponentTag*

  #include <base_lco_with_value.hpp> The base_lco<void> specialization is used whenever the set_event action for a particular LCO doesn't carry any argument.

  Template Parameters void – This specialization expects no result value and is almost completely equivalent to the plain base_lco.

Public Types

using **wrapping_type** = typename detail::base_lco_wrapping_type<*ComponentTag*, base_lco_with_value>::type

using **base_type_holder** = base_lco_with_value

using **set_value_action** = typename *base_lco*::set_event_action

Public Functions

inline void **get_value**()

Protected Functions

~base_lco_with_value() override = default
  Destructor, needs to be virtual to allow for clean destruction of derived objects

namespace **traits**

hpx::distributed::promise

Defined in header hpx/future.hpp[778].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **distributed**

template<typename **Result**, typename **RemoteResult**>

class **promise**

#include <promise.hpp> A promise can be used by a single *thread* to invoke a (remote) action and wait for the result. The result is expected to be sent back to the promise using the LCO's set_event action

A promise is one of the simplest synchronization primitives provided by HPX. It allows to synchronize on a eager evaluated remote operation returning a result of the type *Result*. The *promise* allows to synchronize exactly one *thread* (the one passed during construction time).

```
// Create the promise (the expected result is a id_type)
hpx::distributed::promise<hpx::id_type> p;

// Get the associated future
future<hpx::id_type> f = p.get_future();

// initiate the action supplying the promise as a
// continuation
apply<some_action>(new continuation(p.get_id()), ...);

// Wait for the result to be returned, yielding control
// in the meantime.
hpx::id_type result = f.get();
// ...
```

Note: The action executed by the promise must return a value of a type convertible to the type as specified by the template parameter *RemoteResult*

Template Parameters

- **Result** – The template parameter *Result* defines the type this promise is expected to return from *promise::get*.
- **RemoteResult** – The template parameter *RemoteResult* defines the type this promise is expected to receive from the remote action.

namespace **lcos**

[778] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/future.hpp

Typedefs

using **instead** = *hpx*::*distributed*::*promise*<Result, RemoteResult>

template<typename **Result**, typename **RemoteResult**, typename **ComponentTag**>

class **base_lco_with_value** : public *hpx*::*lcos*::*base_lco*, public *ComponentTag*

#include <base_lco_with_value.hpp>

template<typename **Action**, typename **Result** = typename *traits*::promise_local_result<typename *Action*::remote_result_type>::type, bool **DirectExecute** = *Action*::direct_execution::value>
class **packaged_action**

#include <packaged_action.hpp> A *packaged_action* can be used by a single *thread* to invoke a (remote) action and wait for the result. The result is expected to be sent back to the *packaged_action* using the LCO's set_event action

A *packaged_action* is one of the simplest synchronization primitives provided by HPX. It allows to synchronize on a eager evaluated remote operation returning a result of the type *Result*.

Note: The action executed using the *packaged_action* as a continuation must return a value of a type convertible to the type as specified by the template parameter *Result*.

Template Parameters
- **Action** – The template parameter *Action* defines the action to be executed by this *packaged_action* instance. The arguments *arg0,… argN* are used as parameters for this action.
- **Result** – The template parameter *Result* defines the type this *packaged_action* is expected to return from its associated future *packaged_action::get_future*.
- **DirectExecute** – The template parameter *DirectExecute* is an optimization aid allowing to execute the action directly if the target is local (without spawning a new thread for this). This template does not have to be supplied explicitly as it is derived from the template parameter *Action*.

namespace **lcos**

hpx/async_distributed/packaged_action.hpp

Defined in header hpx/async_distributed/packaged_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **lcos**

template<typename **Action**, typename **Result** = typename *traits*::promise_local_result<typename *Action*::remote_result_type>::type, bool **DirectExecute** = *Action*::direct_execution::value>
class **packaged_action**

#include <packaged_action.hpp>

template<typename **Action**, typename **Result**>

class **packaged_action**<*Action*, *Result*, false> : public *hpx*::*distributed*::*promise*<*Result*, *hpx*::*traits*::extract_action<*Action*>::remote_result_type>

Subclassed by *hpx::lcos::packaged_action< Action, Result, true >*

Public Functions

inline **packaged_action**()

template<typename **Allocator**>
inline **packaged_action**(*std*::allocator_arg_t, *Allocator* const &alloc)

template<typename ...**Ts**>
inline void **post**(*hpx*::id_type const &id, *Ts*&&... vs)

template<typename ...**Ts**>
inline void **post**(*naming*::address &&addr, *hpx*::id_type const &id, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **post_cb**(*hpx*::id_type const &id, *Callback* &&cb, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **post_cb**(*naming*::address &&addr, *hpx*::id_type const &id, *Callback* &&cb, *Ts*&&... vs)

template<typename ...**Ts**>
inline void **post_p**(*hpx*::id_type const &id, *hpx*::*launch* policy, *Ts*&&... vs)

template<typename ...**Ts**>
inline void **post_p**(*naming*::address &&addr, *hpx*::id_type const &id, *hpx*::*launch* policy, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **post_p_cb**(*hpx*::id_type const &id, *hpx*::*launch* policy, *Callback* &&cb, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **post_p_cb**(*naming*::address &&addr, *hpx*::id_type const &id, *hpx*::*launch* policy, *Callback* &&cb, *Ts*&&... vs)

template<typename ...**Ts**>
inline void **post_deferred**(*naming*::address &&addr, *hpx*::id_type const &id, *hpx*::*launch* policy, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **post_deferred_cb**(*naming*::address &&addr, *hpx*::id_type const &id, *hpx*::*launch* policy, *Callback* &&cb, *Ts*&&... vs)

Protected Types

using **action_type** = typename *hpx*::*traits*::extract_action<*Action*>::type

using **remote_result_type** = typename *action_type*::remote_result_type

using **base_type** = *hpx*::*distributed*::*promise*<*Result*, *remote_result_type*>

Protected Functions

template<typename ...**Ts**>
inline void **do_post**(*naming*::address &&addr, *hpx*::id_type const &id, *hpx*::*launch* policy, *Ts*&&... vs)

template<typename ...**Ts**>
inline void **do_post**(*hpx*::id_type const &id, *hpx*::*launch* policy, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **do_post_cb**(*naming*::address &&addr, *hpx*::id_type const &id, *hpx*::*launch* policy, *Callback* &&cb, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **do_post_cb**(*hpx*::id_type const &id, *hpx*::*launch* policy, *Callback* &&cb, *Ts*&&... vs)

template<typename **Action**, typename **Result**>

class **packaged_action**<*Action*, *Result*, true> : public *hpx*::*lcos*::*packaged_action*<*Action*, *Result*, false>

Public Functions

inline **packaged_action**()

Construct a (non-functional) instance of an packaged_action. To use this instance its member function *post* needs to be directly called.

template<typename **Allocator**>
inline **packaged_action**(*std*::allocator_arg_t, *Allocator* const &alloc)

template<typename ...**Ts**>
inline void **post**(*hpx*::id_type const &id, *Ts*&&... vs)

template<typename ...**Ts**>
inline void **post**(*naming*::address &&addr, *hpx*::id_type const &id, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **post_cb**(*hpx*::id_type const &id, *Callback* &&cb, *Ts*&&... vs)

template<typename **Callback**, typename ...**Ts**>
inline void **post_cb**(*naming*::address &&addr, *hpx*::id_type const &id, *Callback* &&cb, *Ts*&&... vs)

Private Types

using **action_type** = typename *packaged_action*<*Action*, *Result*, false>::action_type

hpx/async_distributed/promise.hpp

Defined in header hpx/async_distributed/promise.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<>

class *hpx*::*distributed*::**promise**<void, *hpx*::*util*::unused_type> : public *lcos*::detail::promise_base<void, *hpx*::*util*::unused_type, *lcos*::detail::promise_data<void>>

Public Functions

promise() = default

constructs a promise object and a shared state.

template<typename **Allocator**>
inline **promise**(*std*::allocator_arg_t, *Allocator* const &a)

constructs a promise object and a shared state. The constructor uses the allocator a to allocate the memory for the shared state.

promise(*promise* &&other) noexcept = default

constructs a new promise object and transfers ownership of the shared state of other (if any) to the newly-constructed object.

Post other has no shared state.

~promise() = default

Abandons any shared state.

promise &**operator=**(*promise* &&other) noexcept = default

Abandons any shared state (30.6.4) and then as if promise(HPX_MOVE(other)).swap(*this).

Returns *this.

inline void **swap**(*promise* &other) noexcept

Exchanges the shared state of *this and other.

Post *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

inline void **set_value**()

atomically stores the value r in the shared state and makes that state ready (30.6.4).

Throws future_error – if its shared state already has a stored value. if shared state has no stored value exception is raised. promise_already_satisfied if its shared state already has a stored value or exception. no_state if *this has no shared state.

Private Types

using **base_type** = *lcos*::detail::promise_base<void, *hpx*::*util*::unused_type, *lcos*::detail::promise_data<void>>

template<typename **R**, typename **Allocator**>

struct **uses_allocator**<*hpx*::*distributed*::*promise*<*R*>, *Allocator*> : public true_type

#include <promise.hpp> Requires: Allocator shall be an allocator (17.6.3.5)

namespace **hpx**

namespace **distributed**

Functions

template<typename **Result**, typename **RemoteResult**>
void **swap**(*promise*<*Result*, *RemoteResult*> &x, *promise*<*Result*, *RemoteResult*> &y) noexcept

template<typename **Result**, typename **RemoteResult**>

class **promise**

#include <promise.hpp>

template<> unused_type > : public lcos::detail::promise_base< void, hpx::util::unused_type, lcos::detail::promise_data< void > >

Public Functions

promise() = default

constructs a promise object and a shared state.

template<typename **Allocator**>
inline **promise**(*std*::allocator_arg_t, *Allocator* const &a)

constructs a promise object and a shared state. The constructor uses the allocator a to allocate the memory for the shared state.

promise(*promise* &&other) noexcept = default

constructs a new promise object and transfers ownership of the shared state of other (if any) to the newly- constructed object.

Post other has no shared state.

~promise() = default

Abandons any shared state.

promise &**operator=**(*promise* &&other) noexcept = default

Abandons any shared state (30.6.4) and then as if promise(HPX_MOVE(other)).swap(*this).

Returns *this.

inline void **swap**(*promise* &other) noexcept

Exchanges the shared state of *this and other.

Post *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

inline void **set_value**()

atomically stores the value r in the shared state and makes that state ready (30.6.4).

Throws `future_error` – if its shared state already has a stored value. if shared state has no stored value exception is raised. promise_already_satisfied if its shared state already has a stored value or exception. no_state if *this has no shared state.

Private Types

using **base_type** = *lcos*::detail::promise_base<void, *hpx*::*util*::unused_type, *lcos*::detail::promise_data<void>>

namespace **std**

template<typename R, typename Allocator> promise< R >, Allocator > : public true_type

#include <promise.hpp> Requires: Allocator shall be an allocator (17.6.3.5)

hpx/async_distributed/transfer_continuation_action.hpp

Defined in header hpx/async_distributed/transfer_continuation_action.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/async_distributed/trigger_lco.hpp

Defined in header hpx/async_distributed/trigger_lco.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

hpx/async_distributed/trigger_lco_fwd.hpp

Defined in header hpx/async_distributed/trigger_lco_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

void **trigger_lco_event**(*hpx*::id_type const &id, *naming*::address &&addr, bool move_credits = true)

Trigger the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should be triggered.
- **addr** – [in] This represents the addr of the LCO which should be triggered.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

inline void **trigger_lco_event**(*hpx*::id_type const &id, bool move_credits = true)

Trigger the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should be triggered.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **trigger_lco_event**(*hpx*::id_type const &id, *naming*::address &&addr, *hpx*::id_type const &cont, bool move_credits = true)

Trigger the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should be triggered.
- **addr** – [in] This represents the addr of the LCO which should be triggered.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

inline void **trigger_lco_event**(*hpx*::id_type const &id, *hpx*::id_type const &cont, bool move_credits = true)

Trigger the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should be triggered.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
void **set_lco_value**(*hpx*::id_type const &id, *naming*::address &&addr, *Result* &&t, bool move_credits = true)

Set the result value for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the given value.
- **addr** – [in] This represents the addr of the LCO which should be triggered.
- **t** – [in] This is the value which should be sent to the LCO.

- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
std::enable_if_t<!*std*::is_same_v<*std*::decay_t<*Result*>, *naming*::address>> **set_lco_value**(*hpx*::id_type const &id, *Result* &&t, bool move_credits = true)

Set the result value for the (managed) LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the given value.
- **t** – [in] This is the value which should be sent to the LCO.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
std::enable_if_t<!*std*::is_same_v<*std*::decay_t<*Result*>, *naming*::address>> **set_lco_value_unmanaged**(*hpx*::id_type const &id, *Result* &&t, bool move_credits = true)

Set the result value for the (unmanaged) LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the given value.
- **t** – [in] This is the value which should be sent to the LCO.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
void **set_lco_value**(*hpx*::id_type const &id, *naming*::address &&addr, *Result* &&t, *hpx*::id_type const &cont, bool move_credits = true)

Set the result value for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the given value.
- **addr** – [in] This represents the addr of the LCO which should be triggered.
- **t** – [in] This is the value which should be sent to the LCO.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>

std::enable_if_t<!std::is_same_v<std::decay_t<Result>, naming::address>> **set_lco_value**(hpx::id_type const &id, Result &&t, hpx::id_type const &cont, bool move_credits = true)

Set the result value for the (managed) LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the given value.
- **t** – [in] This is the value which should be sent to the LCO.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
std::enable_if_t<!std::is_same_v<std::decay_t<Result>, naming::address>> **set_lco_value_unmanaged**(hpx::id_type const &id, Result &&t, hpx::id_type const &cont, bool move_credits = true)

Set the result value for the (unmanaged) LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the given value.
- **t** – [in] This is the value which should be sent to the LCO.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(hpx::id_type const &id, naming::address &&addr, std::exception_ptr const &e, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **addr** – [in] This represents the addr of the LCO which should be triggered.
- **e** – [in] This is the error value which should be sent to the LCO.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*hpx*::id_type const &id, *naming*::address &&addr, *std*::exception_ptr &&e, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **addr** – [in] This represents the addr of the LCO which should be triggered.
- **e** – [in] This is the error value which should be sent to the LCO.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

inline void **set_lco_error**(*hpx*::id_type const &id, *std*::exception_ptr const &e, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **e** – [in] This is the error value which should be sent to the LCO.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

inline void **set_lco_error**(*hpx*::id_type const &id, *std*::exception_ptr &&e, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **e** – [in] This is the error value which should be sent to the LCO.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*hpx*::id_type const &id, *naming*::address &&addr, *std*::exception_ptr const &e, *hpx*::id_type const &cont, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **addr** – [in] This represents the addr of the LCO which should be triggered.
- **e** – [in] This is the error value which should be sent to the LCO.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*hpx*::id_type const &id, *naming*::address &&addr, *std*::exception_ptr &&e, *hpx*::id_type const &cont, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **addr** – [in] This represents the addr of the LCO which should be triggered.

- **e** – [in] This is the error value which should be sent to the LCO.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

inline void **set_lco_error**(*hpx*::id_type const &id, *std*::exception_ptr const &e, *hpx*::id_type const &cont, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **e** – [in] This is the error value which should be sent to the LCO.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

inline void **set_lco_error**(*hpx*::id_type const &id, *std*::exception_ptr &&e, *hpx*::id_type const &cont, bool move_credits = true)

Set the error state for the LCO referenced by the given id.

Parameters

- **id** – [in] This represents the id of the LCO which should receive the error value.
- **e** – [in] This is the error value which should be sent to the LCO.
- **cont** – [in] This represents the LCO to trigger after completion.
- **move_credits** – [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

checkpoint

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/checkpoint/checkpoint.hpp

Defined in header hpx/checkpoint/checkpoint.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

This header defines the save_checkpoint and restore_checkpoint functions. These functions are designed to help HPX application developer's checkpoint their applications. Save_checkpoint serializes one or more objects and saves them as a byte stream. Restore_checkpoint converts the byte stream back into instances of the objects.

namespace **hpx**

namespace **util**

Functions

inline *std*::ostream &**operator<<**(*std*::ostream &ost, *checkpoint* const &ckp)

Operator<< Overload

This overload is the main way to write data from a checkpoint to an object such as a file. Inside the function, the size of the checkpoint will be written to the stream before the checkpoint's data. The operator>> overload uses this to read the correct number of bytes. Be mindful of this additional write and read when you use different facilities to write out or read in data to a checkpoint!

Parameters

- **ost** – Output stream to write to.
- **ckp** – Checkpoint to copy from.

Returns Operator<< returns the ostream object.

inline *std*::istream &**operator>>**(*std*::istream &ist, *checkpoint* &ckp)

Operator>> Overload

This overload is the main way to read in data from an object such as a file to a checkpoint. It is important to note that inside the function, the first variable to be read is the size of the checkpoint. This size variable is written to the stream before the checkpoint's data in the operator<< overload. Be mindful of this additional read and write when you use different facilities to read in or write out data from a checkpoint!

Parameters

- **ist** – Input stream to write from.
- **ckp** – Checkpoint to write to.

Returns Operator>> returns the ostream object.

template<typename **T**, typename ...**Ts**, typename **U** = typename *std*::enable_if<!*hpx*::*traits*::is_launch_policy<*T*>::value && !*std*::is_same<typename *std*::decay<*T*>::type, *checkpoint*>::value>::type>
hpx::*future*<*checkpoint*> **save_checkpoint**(*T* &&t, *Ts*&&... ts)

Save_checkpoint

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- **T** – Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **Ts** – More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **U** – This parameter is used to make sure that T is not a launch policy or a checkpoint. This forces the compiler to choose the correct overload.

Parameters

- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a checkpoint.

template<typename **T**, typename ...**Ts**>
hpx::*future*<*checkpoint*> **save_checkpoint**(*checkpoint* &&c, *T* &&t, *Ts*&&... ts)

Save_checkpoint - Take a pre-initialized checkpoint

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- **T** – Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **Ts** – More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- **c** – Takes a pre-initialized checkpoint to copy data into.
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a checkpoint.

template<typename **T**, typename ...**Ts**, typename **U** = typename *std*::enable_if<!*std*::is_same<typename *std*::decay<*T*>::type, *checkpoint*>::value>::type>
hpx::*future*<*checkpoint*> **save_checkpoint**(*hpx*::*launch* p, *T* &&t, *Ts*&&... ts)

Save_checkpoint - Policy overload

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- **T** – Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **Ts** – More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- **p** – Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a checkpoint.

template<typename **T**, typename ...**Ts**>
hpx::*future*<*checkpoint*> **save_checkpoint**(*hpx*::*launch* p, *checkpoint* &&c, *T* &&t, *Ts*&&... ts)

Save_checkpoint - Policy overload & pre-initialized checkpoint

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- **T** – Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **Ts** – More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- **p** – Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- **c** – Takes a pre-initialized checkpoint to copy data into.
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Save_checkpoint returns a future to a checkpoint with one exception: if you pass hpx::launch::sync as the first argument. In this case save_checkpoint will simply return a checkpoint.

template<typename **T**, typename ...**Ts**, typename **U** = typename *std*::enable_if<!*std*::is_same<typename *std*::decay<*T*>::type, *checkpoint*>::value>::type>
checkpoint **save_checkpoint**(*hpx*::*launch*::sync_policy sync_p, *T* &&t, *Ts*&&... ts)

Save_checkpoint - Sync_policy overload

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- **T** – Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **Ts** – More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **U** – This parameter is used to make sure that T is not a checkpoint. This forces the compiler to choose the correct overload.

Parameters

- **sync_p** – hpx::launch::sync_policy
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Save_checkpoint which is passed hpx::launch::sync_policy will return a checkpoint which contains the serialized values checkpoint.

template<typename **T**, typename ...**Ts**>
checkpoint **save_checkpoint**(*hpx*::*launch*::sync_policy sync_p, *checkpoint* &&c, *T* &&t, *Ts*&&... ts)

Save_checkpoint - Sync_policy overload & pre-init. checkpoint

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- **T** – Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- **Ts** – More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- **sync_p** – hpx::launch::sync_policy
- **c** – Takes a pre-initialized checkpoint to copy data into.
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Save_checkpoint which is passed hpx::launch::sync_policy will return a checkpoint which contains the serialized values checkpoint.

template<typename **T**, typename ...**Ts**, typename **U** = typename *std*::enable_if<!*hpx*::*traits*::is_launch_policy<*T*>::value && !*std*::is_same<typename *std*::decay<*T*>::type, *checkpoint*>::value>::type>
hpx::*future*<*checkpoint*> **prepare_checkpoint**(*T* const &t, *Ts* const&... ts)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Template Parameters

- **T** – A container to restore.
- **Ts** – Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

template<typename **T**, typename ...**Ts**>
hpx::*future*<*checkpoint*> **prepare_checkpoint**(*checkpoint* &&c, *T* const &t, *Ts* const&... ts)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Template Parameters

- **T** – A container to restore.
- **Ts** – Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- **c** – Takes a pre-initialized checkpoint to prepare
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

template<typename **T**, typename ...**Ts**, typename **U** = typename *std*::enable_if<!*std*::is_same<*T*, *checkpoint*>::value>::type>
hpx::*future*<*checkpoint*> **prepare_checkpoint**(*hpx*::*launch* p, *T* const &t, *Ts* const&... ts)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Template Parameters

- **T** – A container to restore.
- **Ts** – Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- **p** – Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

template<typename **T**, typename ...**Ts**>
hpx::*future*<*checkpoint*> **prepare_checkpoint**(*hpx*::*launch* p, *checkpoint* &&c, *T* const &t, *Ts* const&... ts)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

Template Parameters

- **T** – A container to restore.
- **Ts** – Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- **p** – Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- **c** – Takes a pre-initialized checkpoint to prepare
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

template<typename **T**, typename ...**Ts**>
void **restore_checkpoint**(*checkpoint* const &c, *T* &t, *Ts*&... ts)

Restore_checkpoint

Restore_checkpoint takes a checkpoint object as a first argument and the containers which will be filled from the byte stream (in the same order as they were placed in save_checkpoint). Restore_checkpoint can resurrect a stored component in two ways: by passing in a instance of a component's shared_ptr or by passing in an instance of the component's client.

Template Parameters

- **T** – A container to restore.

- **Ts** – Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters

- **c** – The checkpoint to restore.
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Restore_checkpoint returns void.

class **checkpoint**

#include <checkpoint.hpp> Checkpoint Object

Checkpoint is the container object which is produced by save_checkpoint and is consumed by a restore_checkpoint. A checkpoint may be moved into the save_checkpoint object to write the byte stream to the pre-created checkpoint object.

Checkpoints are able to store all containers which are able to be serialized including components.

Public Types

using **const_iterator** = *std*::vector<char>::const_iterator

Public Functions

checkpoint() = default

~checkpoint() = default

checkpoint(*checkpoint* const &c) = default

checkpoint(*checkpoint* &&c) noexcept = default

inline **checkpoint**(*std*::vector<char> const &vec)

inline **checkpoint**(*std*::vector<char> &&vec) noexcept

checkpoint &**operator=**(*checkpoint* const &c) = default

checkpoint &**operator=**(*checkpoint* &&c) noexcept = default

inline *const_iterator* **begin**() const noexcept

inline *const_iterator* **end**() const noexcept

inline *std*::size_t **size**() const noexcept

inline char ***data**() noexcept

inline char const ***data**() const noexcept

Private Functions

template<typename **Archive**>
inline void **serialize**(*Archive* &arch, const unsigned int)

Private Members

std::vector<char> **data_**

Friends

friend class hpx::serialization::access

friend *std*::ostream &**operator<<**(*std*::ostream &ost, *checkpoint* const &ckp)

Operator<< Overload

This overload is the main way to write data from a checkpoint to an object such as a file. Inside the function, the size of the checkpoint will be written to the stream before the checkpoint's data. The operator>> overload uses this to read the correct number of bytes. Be mindful of this additional write and read when you use different facilities to write out or read in data to a checkpoint!

Parameters

- **ost** – Output stream to write to.
- **ckp** – Checkpoint to copy from.

Returns Operator<< returns the ostream object.

friend *std*::istream &**operator>>**(*std*::istream &ist, *checkpoint* &ckp)

Operator>> Overload

This overload is the main way to read in data from an object such as a file to a checkpoint. It is important to note that inside the function, the first variable to be read is the size of the checkpoint. This size variable is written to the stream before the checkpoint's data in the operator<< overload. Be mindful of this additional read and write when you use different facilities to read in or write out data from a checkpoint!

Parameters

- **ist** – Input stream to write from.
- **ckp** – Checkpoint to write to.

Returns Operator>> returns the ostream object.

template<typename **T**, typename ...**Ts**>
friend void **restore_checkpoint**(*checkpoint* const &c, *T* &t, *Ts*&... ts)

Restore_checkpoint

Restore_checkpoint takes a checkpoint object as a first argument and the containers which will be filled from the byte stream (in the same order as they were placed in save_checkpoint). Restore_checkpoint can resurrect a stored component in two ways: by passing in a instance of a component's shared_ptr or by passing in an instance of the component's client.

Template Parameters

- **T** – A container to restore.
- **Ts** – Other containers to restore. Containers must be in the same order that they were inserted into the checkpoint.

Parameters
- **c** – The checkpoint to restore.
- **t** – A container to restore.
- **ts** – Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Returns Restore_checkpoint returns void.

inline friend bool **operator==**(*checkpoint* const &lhs, *checkpoint* const &rhs)

inline friend bool **operator!=**(*checkpoint* const &lhs, *checkpoint* const &rhs)

checkpoint_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/checkpoint_base/checkpoint_data.hpp

Defined in header hpx/checkpoint_base/checkpoint_data.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **util**

Functions

template<typename **Container**, typename ...**Ts**>
void **save_checkpoint_data**(*Container* &data, *Ts*&&... ts)

save_checkpoint_data

Save_checkpoint_data takes any number of objects which a user may wish to store in the given container.

Template Parameters
- **Container** – Container used to store the check-pointed data.
- **Ts** – Types of variables to checkpoint

Parameters
- **data** – Container instance used to store the checkpoint data
- **ts** – Variable instances to be inserted into the checkpoint.

template<typename ...**Ts**>
std::size_t **prepare_checkpoint_data**(*Ts* const&... ts)

prepare_checkpoint_data

prepare_checkpoint_data takes any number of objects which a user may wish to store in a subsequent save_checkpoint_data operation. The function will return the number of bytes necessary to store the data that will be produced.

Template Parameters **Ts** – Types of variables to checkpoint

Parameters **ts** – Variable instances to be inserted into the checkpoint.

template<typename **Container**, typename ...**Ts**>
void **restore_checkpoint_data**(*Container* const &cont, *Ts*&... ts)

restore_checkpoint_data

restore_checkpoint_data takes any number of objects which a user may wish to restore from the given container. The sequence of objects has to correspond to the sequence of objects for the corresponding call to save_checkpoint_data that had used the given container instance.

Template Parameters

- **Container** – Container used to restore the check-pointed data.
- **Ts** – Types of variables to restore

Parameters

- **cont** – Container instance used to restore the checkpoint data
- **ts** – Variable instances to be restored from the container

struct **checkpointing_tag**

template<>

struct **extra_data_helper**<*checkpointing_tag*>

Public Static Functions

static extra_data_id_type **id**() noexcept

static inline constexpr void **reset**(*checkpointing_tag**) noexcept

collectives

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/collectives/all_gather.hpp

Defined in header hpx/collectives/all_gather.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

template<typename **T**>
hpx::*future*<*std*::vector<*std*::decay_t<*T*>>> **all_gather**(char const *basename, *T* &&result, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

AllGather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the all_gather operation
- **local_result** – The value to transmit to all participating sites from this call site.
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero. \params root_site The site that is responsible for creating the all_gather support object. This value is optional and defaults to '0' (zero).

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_gather operation has been completed.

template<typename **T**>
hpx::*future*<*std*::vector<*std*::decay_t<*T*>>> **all_gather**(*communicator* comm, *T* &&result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

AllGather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

AllGather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater

than zero.

- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_gather operation has been completed.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_gather operation has been completed.

hpx/collectives/all_reduce.hpp

Defined in header hpx/collectives/all_reduce.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

template<typename **T**, typename **F**>
hpx::*future*<*std*::decay_t<*T*>> **all_reduce**(char const *basename, *T* &&result, *F* &&op, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

AllReduce a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the all_reduce operation
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **num_sites** – The number of participating sites (default: all localities).
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns. \params root_site The site that is responsible for creating the all_reduce support object. This value is optional and defaults to '0' (zero).

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_reduce operation has been completed.

template<typename **T**, typename **F**>
hpx::*future*<*std*::decay_t<*T*>> **all_reduce**(*communicator* comm, *T* &&result, *F* &&op, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

AllReduce a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

AllReduce a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_reduce operation has been completed.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_reduce operation has been completed.

hpx/collectives/all_to_all.hpp

Defined in header hpx/collectives/all_to_all.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

template<typename **T**>
hpx::*future*<*std*::vector<*std*::decay_t<*T*>>> **all_to_all**(char const *basename, *T* &&result, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

AllToAll a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the all_to_all operation
- **local_result** – The value to transmit to all participating sites from this call site.
- **num_sites** – The number of participating sites (default: all localities).
- **generation** – The generational counter identifying the sequence number of the all_to_all operation performed on the given base name. This is optional and needs to be supplied only if the all_to_all operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns. \params root_site The site that is responsible for creating the all_to_all support object. This value is optional and defaults to '0' (zero).

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_to_all operation has been completed.

template<typename **T**>
hpx::*future*<*std*::vector<*std*::decay_t<*T*>>> **all_to_all**(*communicator* comm, *T* &&result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

AllToAll a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

AllToAll a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_to_all operation performed on the given base name. This is optional and needs to be supplied only if the all_to_all operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **generation** – The generational counter identifying the sequence number of the all_to_all operation performed on the given base name. This is optional and needs to be supplied only if the all_to_all operation on the given base name has to be performed

more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_to_all operation has been completed.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_to_all operation has been completed.

hpx/collectives/argument_types.hpp

Defined in header hpx/collectives/argument_types.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Typedefs

using **num_sites_arg** = detail::argument_type<detail::num_sites_tag>

The number of participating sites (default: all localities)

using **this_site_arg** = detail::argument_type<detail::this_site_tag>

The local end of the communication channel.

using **that_site_arg** = detail::argument_type<detail::that_site_tag>

The opposite end of the communication channel.

using **generation_arg** = detail::argument_type<detail::generation_tag>

The generational counter identifying the sequence number of the operation performed on the given base name. It needs to be supplied only if the operation on the given base name has to be performed more than once. It must be a positive number greater than zero.

using **root_site_arg** = detail::argument_type<detail::root_site_tag, 0>

The site that is responsible for creating the support object of the operation. It defaults to '0' (zero).

using **tag_arg** = detail::argument_type<detail::tag_tag, 0>

The tag identifying the concrete operation.

using **arity_arg** = detail::argument_type<detail::arity_tag>

The number of children each of the communication nodes is connected to (default: picked based on num_sites).

hpx::distributed::barrier

Defined in header hpx/barrier.hpp[779].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **distributed**

Functions

explicit **barrier**(*std*::string const &base_name)

Creates a barrier, rank is locality id, size is number of localities

A barrier *base_name* is created. It expects that hpx::get_num_localities() participate and the local rank is hpx::get_locality_id().

Parameters **base_name** – The name of the barrier

barrier(*std*::string const &base_name, *std*::size_t num)

Creates a barrier with a given size, rank is locality id

A barrier *base_name* is created. It expects that *num* participate and the local rank is hpx::get_locality_id().

Parameters

- **base_name** – The name of the barrier
- **num** – The number of participating threads

barrier(*std*::string const &base_name, *std*::size_t num, *std*::size_t rank)

Creates a barrier with a given size and rank

A barrier *base_name* is created. It expects that *num* participate and the local rank is *rank*.

Parameters

- **base_name** – The name of the barrier
- **num** – The number of participating threads
- **rank** – The rank of the calling site for this invocation

barrier(*std*::string const &base_name, *std*::vector<*std*::size_t> const &ranks, *std*::size_t rank)

Creates a barrier with a vector of ranks

A barrier *base_name* is created. It expects that ranks.size() and the local rank is *rank* (must be contained in *ranks*).

Parameters

- **base_name** – The name of the barrier
- **ranks** – Gives a list of participating ranks (this could be derived from a list of locality ids

[779] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/barrier.hpp

- **rank** – The rank of the calling site for this invocation

void **wait**() const

Wait until each participant entered the barrier. Must be called by all participants

Returns This function returns once all participants have entered the barrier (have called *wait*).

hpx::*future*<void> **wait**(*hpx*::*launch*::async_policy) const

Wait until each participant entered the barrier. Must be called by all participants

Returns a future that becomes ready once all participants have entered the barrier (have called *wait*).

static void **synchronize**()

Perform a global synchronization using the default global barrier The barrier is created once at startup and can be reused throughout the lifetime of an HPX application.

Note: This function currently does not support dynamic connection and disconnection of localities.

hpx/collectives/broadcast.hpp

Defined in header hpx/collectives/broadcast.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

template<typename **T**>
hpx::*future*<void> **broadcast_to**(char const *basename, *T* &&local_result, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Broadcast a value to different call sites

This function sends a set of values to all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the broadcast operation
- **local_result** – A value to transmit to all participating sites from this call site.
- **num_sites** – The number of participating sites (default: all localities).
- **generation** – The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future that will become ready once the broadcast operation has been completed.

template<typename **T**>

hpx::*future*<void> **broadcast_to**(*communicator* comm, *T* &&local_result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Broadcast a value to different call sites

This function sends a set of values to all call sites operating on the given base name.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – A value to transmit to all participating sites from this call site.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future that will become ready once the broadcast operation has been completed.

template<typename **T**>
hpx::*future*<void> **broadcast_to**(*communicator* comm, *generation_arg* generation, *T* &&local_result, *this_site_arg* this_site = *this_site_arg*())

Broadcast a value to different call sites

This function sends a set of values to all call sites operating on the given base name.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – A value to transmit to all participating sites from this call site.
- **generation** – The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future that will become ready once the broadcast operation has been completed.

template<typename **T**>
hpx::*future*<*T*> **broadcast_from**(char const *basename, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Receive a value that was broadcast to different call sites

This function sends a set of values to all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the broadcast operation
- **generation** – The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.

- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding the value that was sent to all participating sites. It will become ready once the broadcast operation has been completed.

template<typename **T**>
hpx::*future*<*T*> **broadcast_from**(*communicator* comm, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Receive a value that was broadcast to different call sites

This function sends a set of values to all call sites operating on the given base name.

Receive a value that was broadcast to different call sites

This function sends a set of values to all call sites operating on the given base name.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Note: The generation values from corresponding *broadcast_to* and *broadcast_from* have to match.

Parameters
- **comm** – A communicator object returned from *create_communicator*
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **generation** – The generational counter identifying the sequence number of the broadcast operation performed on the given base name. This is optional and needs to be supplied only if the broadcast operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding the value that was sent to all participating sites. It will become ready once the broadcast operation has been completed.

Returns This function returns a future holding the value that was sent to all participating sites. It will become ready once the broadcast operation has been completed.

hpx/collectives/broadcast_direct.hpp

Defined in header hpx/collectives/broadcast_direct.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **lcos**

Functions

template<typename Action, typename ArgN, ...
> hpx::future< std::vector< decltype(Action(hpx::id_type, ArgN,...
))> > broadcast (std::vector< hpx::id_type > const &ids, ArgN argN,...)

Perform a distributed broadcast operation.

The function hpx::lcos::broadcast performs a distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

Note: If decltype(Action(…)) is void, then the result of this function is future<void>.

Parameters

- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **argN** – [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall reduction operation.

template<typename Action, typename ArgN, ...
> void broadcast_post (std::vector< hpx::id_type > const &ids, ArgN argN,...)

Perform an asynchronous (fire&forget) distributed broadcast operation.

The function hpx::lcos::broadcast_post performs an asynchronous (fire&forget) distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

Parameters

- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **argN** – [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

template<typename Action, typename ArgN, ...
> hpx::future< std::vector< decltype(Action(hpx::id_type, ArgN,...,
std::size_t))> > broadcast_with_index (std::vector< hpx::id_type > const &ids,
ArgN argN,...)

Perform a distributed broadcast operation.

The function hpx::lcos::broadcast_with_index performs a distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note: If decltype(Action(…)) is void, then the result of this function is future<void>.

Parameters
- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **argN** – [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall reduction operation.

```
template<typename Action, typename ArgN, ...
> void broadcast_post_with_index (std::vector< hpx::id_type > const &ids,
ArgN argN,...)
```

Perform an asynchronous (fire&forget) distributed broadcast operation.

The function hpx::lcos::broadcast_post_with_index performs an asynchronous (fire&forget) distributed broadcast operation resulting in action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The given action is invoked asynchronously on all given identifiers, and the arguments ArgN are passed along to those invocations.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Parameters
- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **argN** – [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

hpx/collectives/channel_communicator.hpp

Defined in header hpx/collectives/channel_communicator.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

hpx::*future*<*channel_communicator*> **create_channel_communicator**(char const *basename, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*())

Create a new communicator object usable with peer-to-peer channel-based operations

This functions creates a new communicator object that can be called in order to pre-allocate a communicator object usable with multiple invocations of channel-based peer-to-peer operations.

Parameters

- **basename** – The base name identifying the collective operation
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future to a new communicator object usable with the collective operation.

channel_communicator **create_channel_communicator**(*hpx*::*launch*::sync_policy, char const *basename, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*())

Create a new communicator object usable with peer-to-peer channel-based operations

This functions creates a new communicator object that can be called in order to pre-allocate a communicator object usable with multiple invocations of channel-based peer-to-peer operations.

Parameters

- **basename** – The base name identifying the collective operation
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a new communicator object usable with the collective operation.

template<typename **T**>
hpx::*future*<void> **set**(*channel_communicator* comm, *that_site_arg* site, *T* &&value, *tag_arg* tag = *tag_arg*())

Send a value to the given site

This function sends a value to the given site based on the given communicator.

Parameters

- **comm** – The channel communicator object to use for the data transfer
- **site** – The destination site
- **value** – The value to send
- **tag** – The (optional) tag identifying the concrete operation

Returns This function returns a future<void> that becomes ready once the data transfer operation has finished.

template<typename **T**>
hpx::*future*<*T*> **get**(*channel_communicator* comm, *that_site_arg* site, *tag_arg* tag = *tag_arg*())

Send a value to the given site

This function receives a value from the given site based on the given communicator.

Parameters

- **comm** – The channel communicator object to use for the data transfer
- **site** – The source site

Returns This function returns a future<T> that becomes ready once the data transfer operation has finished. The future will hold the received value.

class **channel_communicator**

#include <channel_communicator.hpp> A handle identifying the communication channel to use for get/set operations

hpx/collectives/communication_set.hpp

Defined in header hpx/collectives/communication_set.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

communicator **create_communication_set**(char const *basename, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *arity_arg* arity = *arity_arg*())

The function *create_communication_set* sets up a (distributed) tree-like communication structure that can be used with any of the collective APIs (such like *all_to_all* and similar).

Parameters

- **basename** – The base name identifying the all_to_all operation
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the collective operation performed on the given base name. This is optional and needs to be supplied only if the collective operation on the given base name has to be performed more than once.
- **arity** – The number of children each of the communication nodes is connected to (default: picked based on num_sites).

Returns This function returns a new communicator object usable with the collective operation.

hpx/collectives/create_communicator.hpp

Defined in header hpx/collectives/create_communicator.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

communicator **create_communicator**(char const *basename, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

Create a new communicator object usable with any collective operation

This functions creates a new communicator object that can be called in order to pre-allocate a communicator object usable with multiple invocations of any of the collective operations (such as *all_gather*, *all_reduce*, *all_to_all*, *broadcast*, etc.).

Parameters

- **basename** – The base name identifying the collective operation
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the collective operation performed on the given base name. This is optional and needs to be supplied only if the collective operation on the given base name has to be performed more than once.
- **root_site** – The site that is responsible for creating the collective support object. This value is optional and defaults to '0' (zero).

Returns This function returns a new communicator object usable with the collective operation.

communicator **create_local_communicator**(char const *basename, *num_sites_arg* num_sites, *this_site_arg* this_site, *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

Create a new communicator object usable with any local collective operation

This functions creates a new communicator object that can be called in order to pre-allocate a communicator object usable with multiple invocations of any of the collective operations (such as *all_gather*, *all_reduce*, *all_to_all*, *broadcast*, etc.).

Parameters

- **basename** – The base name identifying the collective operation
- **num_sites** – The number of participating sites
- **this_site** – The sequence number of this invocation (usually the sequence number of the object participating in the collective operation). This value must be in the range [0, num_sites).

- **generation** – The generational counter identifying the sequence number of the collective operation performed on the given base name. This is optional and needs to be supplied only if the collective operation on the given base name has to be performed more than once.
- **root_site** – The site that is responsible for creating the collective support object. This value is optional and defaults to '0' (zero).

Returns This function returns a new communicator object usable for all local collective operations.

struct **communicator**

#include <create_communicator.hpp> A communicator instance represents the list of sites that participate in a particular collective operation.

Public Functions

void **set_info**(*num_sites_arg* num_sites, *this_site_arg* this_site) noexcept

Store the number of used sites and the index of the current site for this communicator instance.

Parameters

- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this site (usually the locality id).

std::pair<*num_sites_arg*, *this_site_arg*> **get_info**() const noexcept

Retrieve the number of used sites and the index of the current site for this communicator instance.

bool **is_root**() const

Return whether this communicator instance represents the root site of the communication operation.

hpx/collectives/exclusive_scan.hpp

Defined in header hpx/collectives/exclusive_scan.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

template<typename **T**, typename **F**>
hpx::*future*<*std*::decay_t<*T*>> **exclusive_scan**(char const *basename, *T* &&result, *F* &&op, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

Exclusive scan a set of values from different call sites

This function performs an exclusive scan operation on a set of values received from all call sites operating on the given base name.

Note: The result returned on the root_site is always the same as the result returned on thus_site == 1 and is the same as the value provided by the root_site.

Parameters
- **basename** – The base name identifying the exclusive_scan operation
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the exclusive_scan operation performed on the given base name. This is optional and needs to be supplied only if the exclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero. \params root_site The site that is responsible for creating the exclusive_scan support object. This value is optional and defaults to '0' (zero).

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the exclusive_scan operation has been completed.

template<typename **T**, typename **F**>
hpx::*future*<*std*::decay_t<*T*>> **exclusive_scan**(*communicator* comm, *T* &&result, *F* &&op, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Exclusive scan a set of values from different call sites

This function performs an exclusive scan operation on a set of values received from all call sites operating on the given base name.

Exclusive scan a set of values from different call sites

This function performs an exclusive scan operation on a set of values received from all call sites operating on the given base name.

Note: The result returned on the root_site is always the same as the result returned on thus_site == 1 and is the same as the value provided by the root_site.

Note: The result returned on the root_site is always the same as the result returned on thus_site == 1 and is the same as the value provided by the root_site.

Parameters
- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the exclusive_scan operation performed on the given base name. This is optional and needs to be supplied only if the exclusive_scan operation on the given base name has to be performed

more than once. The generation number (if given) must be a positive number greater than zero.

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **generation** – The generational counter identifying the sequence number of the exclusive_scan operation performed on the given base name. This is optional and needs to be supplied only if the exclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the exclusive_scan operation has been completed.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the exclusive_scan operation has been completed.

hpx/collectives/fold.hpp

Defined in header hpx/collectives/fold.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **lcos**

Functions

```
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...
> hpx::future< decltype(Action(hpx::id_type, ArgN,...
))> fold (std::vector< hpx::id_type > const &ids, FoldOp &&fold_op, Init &&init,
ArgN argN,...)
```

Perform a distributed fold operation.

The function hpx::lcos::fold performs a distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

Parameters

- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **fold_op** – [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.

- **init** – [in] The initial value to be used for the folding operation
- **argN** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall folding operation.

```
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...
> hpx::future< decltype(Action(hpx::id_type, ArgN,...,
std::size_t))> fold_with_index (std::vector< hpx::id_type > const &ids,
FoldOp &&fold_op, Init &&init, ArgN argN,...)
```

Perform a distributed folding operation.

The function hpx::lcos::fold_with_index performs a distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

Parameters

- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **fold_op** – [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- **init** – [in] The initial value to be used for the folding operation
- **argN** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall folding operation.

```
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...
> hpx::future< decltype(Action(hpx::id_type, ArgN,...
))> inverse_fold (std::vector< hpx::id_type > const &ids, FoldOp &&fold_op,
Init &&init, ArgN argN,...)
```

Perform a distributed inverse folding operation.

The function hpx::lcos::inverse_fold performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

Parameters

- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **fold_op** – [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.

- **init** – [in] The initial value to be used for the folding operation
- **argN** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall folding operation.

```
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...
> hpx::future< decltype(Action(hpx::id_type, ArgN,...,
std::size_t))> inverse_fold_with_index (std::vector< hpx::id_type > const &ids,
FoldOp &&fold_op, Init &&init, ArgN argN,...)
```

Perform a distributed inverse folding operation.

The function hpx::lcos::inverse_fold_with_index performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note: The type of the initial value must be convertible to the result type returned from the invoked action.

Parameters

- **ids** – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- **fold_op** – [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the folding operation performed on its arguments.
- **init** – [in] The initial value to be used for the folding operation
- **argN** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall folding operation.

hpx/collectives/gather.hpp

Defined in header hpx/collectives/gather.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_here(char const *basename, T &&result,
                                                  num_sites_arg num_sites = num_sites_arg(),
                                                  this_site_arg this_site = this_site_arg(),
                                                  generation_arg generation = generation_arg())
```

Gather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the gather operation
- **result** – The value to transmit to the central gather point from this call site.
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied only if the gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

```
template<typename T>
hpx::future<std::vector<decay_t<T>>> gather_here(communicator comm, T &&result, this_site_arg
                                                  this_site = this_site_arg(), generation_arg
                                                  generation = generation_arg())
```

Gather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Gather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **result** – The value to transmit to the central gather point from this call site.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied only if the gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **result** – The value to transmit to the central gather point from this call site.
- **generation** – The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied

only if the gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

template<typename **T**>
hpx::*future*<*std*::vector<decay_t<*T*>>> **gather_there**(char const *basename, *T* &&result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

Gather a given value at the given call site

This function transmits the value given by *result* to a central gather site (where the corresponding *gather_here* is executed)

Parameters
- **basename** – The base name identifying the gather operation
- **result** – The value to transmit to the central gather point from this call site.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied only if the gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **root_site** – The sequence number of the central gather point (usually the locality id). This value is optional and defaults to 0.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

template<typename **T**>
hpx::*future*<*std*::vector<decay_t<*T*>>> **gather_there**(*communicator* comm, *T* &&result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Gather a given value at the given call site

This function transmits the value given by *result* to a central gather site (where the corresponding *gather_here* is executed)

Gather a given value at the given call site

This function transmits the value given by *result* to a central gather site (where the corresponding *gather_here* is executed)

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Note: The generation values from corresponding *gather_here* and *gather_there* have to match.

Parameters
- **comm** – A communicator object returned from *create_communicator*

- **result** – The value to transmit to the central gather point from this call site.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied only if the gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **result** – The value to transmit to the central gather point from this call site.
- **generation** – The generational counter identifying the sequence number of the gather operation performed on the given base name. This is optional and needs to be supplied only if the gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

Returns This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

hpx/collectives/inclusive_scan.hpp

Defined in header hpx/collectives/inclusive_scan.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

```
template<typename T, typename F>
hpx::future<std::decay_t<T>> inclusive_scan(char const *basename, T &&result, F &&op,
                                            num_sites_arg num_sites = num_sites_arg(),
                                            this_site_arg this_site = this_site_arg(),
                                            generation_arg generation = generation_arg(),
                                            root_site_arg root_site = root_site_arg())
```

Inclusive inclusive_scan a set of values from different call sites

This function performs an inclusive scan operation on a set of values received from all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the inclusive_scan operation
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

- **generation** – The generational counter identifying the sequence number of the inclusive_scan operation performed on the given base name. This is optional and needs to be supplied only if the inclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero. \params root_site The site that is responsible for creating the inclusive_scan support object. This value is optional and defaults to '0' (zero).

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the inclusive_scan operation has been completed.

template<typename **T**, typename **F**>
hpx::*future*<*std*::decay_t<*T*>> **inclusive_scan**(*communicator* comm, *T* &&result, *F* &&op, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Inclusive inclusive_scan a set of values from different call sites

This function performs an inclusive scan operation on a set of values received from all call sites operating on the given base name.

Inclusive inclusive_scan a set of values from different call sites

This function performs an inclusive scan operation on a set of values received from all call sites operating on the given base name.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the inclusive_scan operation performed on the given base name. This is optional and needs to be supplied only if the inclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **local_result** – The value to transmit to all participating sites from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **generation** – The generational counter identifying the sequence number of the inclusive_scan operation performed on the given base name. This is optional and needs to be supplied only if the inclusive_scan operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the inclusive_scan operation has been completed.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the inclusive_scan operation has been completed.

hpx::distributed::latch

Defined in header hpx/latch.hpp[780].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **distributed**

class **latch** : public *components*::client_base<*latch*, *hpx*::*lcos*::server::latch>

#include <latch.hpp> Latch is an implementation of a synchronization primitive that allows multiple threads to wait for a shared event to occur before proceeding. This latch can be invoked in a distributed application.

For a local only latch

See also:

hpx::latch.

Public Functions

latch() = default

explicit **latch**(*std*::ptrdiff_t count)

Initialize the latch

Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

inline **latch**(*hpx*::id_type const &id)

Extension: Create a client side representation for the existing *server::latch* instance with the given global id *id*.

inline **latch**(*hpx*::*future*<*hpx*::id_type> &&f)

Extension: Create a client side representation for the existing *server::latch* instance with the given global id *id*.

inline **latch**(*hpx*::*shared_future*<*hpx*::id_type> const &id)

Extension: Create a client side representation for the existing *server::latch* instance with the given global id *id*.

inline **latch**(*hpx*::*shared_future*<*hpx*::id_type> &&id)

inline void **count_down_and_wait**()

Decrements counter_ by 1 . Blocks at the synchronization point until counter_ reaches 0.

Requires: counter_ > 0.

Synchronization: Synchronizes with all calls that block on this latch and with all is_ready calls on this latch that return true.

Throws `Nothing.` –

[780] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/latch.hpp

inline void **arrive_and_wait**()

Decrements counter_ by update . Blocks at the synchronization point until counter_ reaches 0.

Requires: counter_ > 0.

Synchronization: Synchronizes with all calls that block on this latch and with all is_ready calls on this latch that return true.

Throws `Nothing.` –

inline void **count_down**(*std*::ptrdiff_t n)

Decrements counter_ by n. Does not block.

Requires: counter_ >= n and n >= 0.

Synchronization: Synchronizes with all calls that block on this latch and with all is_ready calls on this latch that return true .

Throws `Nothing.` –

inline bool **is_ready**() const noexcept

Returns: counter_ == 0. Does not block.

Throws `Nothing.` –

inline bool **try_wait**() const noexcept

Returns: counter_ == 0. Does not block.

Throws `Nothing.` –

inline void **wait**() const

If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization point until counter_ reaches 0.

Throws `Nothing.` –

Private Types

typedef *components*::client_base<*latch*, *hpx*::*lcos*::server::latch> **base_type**

namespace **lcos**

hpx/collectives/reduce.hpp

Defined in header hpx/collectives/reduce.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

template<typename **T**, typename **F**>
hpx::*future*<*std*::decay_t<*T*>> **reduce_here**(char const *basename, *T* &&result, *F* &&op, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Reduce a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Parameters

- **basename** – The base name identifying the all_reduce operation
- **local_result** – A value to reduce on the central reduction point from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.

Returns This function returns a future holding a vector with all values send by all participating sites. It will become ready once the all_reduce operation has been completed.

template<typename **T**, typename **F**>
hpx::*future*<decay_t<*T*>> **reduce_here**(*communicator* comm, *T* &&local_result, *F* &&op, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Reduce a set of values from different call sites

This function receives a set of values that are the result of applying a given operator on values supplied from all call sites operating on the given base name.

Reduce a set of values from different call sites

This function receives a set of values that are the result of applying a given operator on values supplied from all call sites operating on the given base name.

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – A value to reduce on the root_site from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed

more than once. The generation number (if given) must be a positive number greater than zero.

- **comm** – A communicator object returned from *create_communicator*
- **local_result** – A value to reduce on the root_site from this call site.
- **op** – Reduction operation to apply to all values supplied from all participating sites
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a value calculated based on the values send by all participating sites. It will become ready once the all_reduce operation has been completed.

Returns This function returns a future holding a value calculated based on the values send by all participating sites. It will become ready once the all_reduce operation has been completed.

template<typename **T**, typename **F**>
hpx::*future*<void> **reduce_there**(char const *basename, *T* &&result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

Reduce a given value at the given call site

This function transmits the value given by *result* to a central reduce site (where the corresponding *reduce_here* is executed)

Parameters

- **basename** – The base name identifying the reduction operation
- **result** – A future referring to the value to transmit to the central reduction point from this call site.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **root_site** – The sequence number of the central reduction point (usually the locality id). This value is optional and defaults to 0.

Returns This function returns a future<void>. It will become ready once the reduction operation has been completed.

template<typename **T**>
hpx::*future*<void> **reduce_there**(*communicator* comm, *T* &&local_result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Reduce a given value at the given call site

This function transmits the value given by *result* to a central reduce site (where the corresponding *reduce_here* is executed)

Reduce a given value at the given call site

This function transmits the value given by *result* to a central reduce site (where the corresponding *reduce_here* is executed)

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

Note: The generation values from corresponding *reduce_here* and *reduce_there* have to match.

Parameters

- `comm` – A communicator object returned from *create_communicator*
- `local_result` – A value to reduce on the central reduction point from this call site.
- `this_site` – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- `generation` – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- `comm` – A communicator object returned from *create_communicator*
- `local_result` – A value to reduce on the central reduction point from this call site.
- `generation` – The generational counter identifying the sequence number of the all_reduce operation performed on the given base name. This is optional and needs to be supplied only if the all_reduce operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- `this_site` – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a value calculated based on the values send by all participating sites. It will become ready once the all_reduce operation has been completed.

Returns This function returns a future holding a value calculated based on the values send by all participating sites. It will become ready once the all_reduce operation has been completed.

hpx/collectives/reduce_direct.hpp

Defined in header hpx/collectives/reduce_direct.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **`hpx`**

Top level HPX namespace.

namespace **`lcos`**

Functions

`template<typename Action, typename ReduceOp, typename ArgN, ...`
`> hpx::future< decltype(Action(hpx::id_type, ArgN,...`
`))> reduce (std::vector< hpx::id_type > const &ids, ReduceOp &&reduce_op,`
`ArgN argN,...)`

Perform a distributed reduction operation.

The function hpx::lcos::reduce performs a distributed reduction operation over results returned from action invocations on a given set of global identifiers. The action can be either a plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

Parameters

- `ids` – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `reduce_op` – [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the reduction operation performed on its arguments.
- `argN` – [in] Any number of arbitrary arguments (passed by by const reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall reduction operation.

`template<typename Action, typename ReduceOp, typename ArgN, ...`
`> hpx::future< decltype(Action(hpx::id_type, ArgN,...,`
`std::size_t))> reduce_with_index (std::vector< hpx::id_type > const &ids,`
`ReduceOp &&reduce_op, ArgN argN,...)`

Perform a distributed reduction operation.

The function hpx::lcos::reduce_with_index performs a distributed reduction operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Parameters

- `ids` – [in] A list of global identifiers identifying the target objects for which the given action will be invoked.
- `reduce_op` – [in] A binary function expecting two results as returned from the action invocations. The function (or function object) is expected to return the result of the reduction operation performed on its arguments.
- `argN` – [in] Any number of arbitrary arguments (passed by by const reference) which will be forwarded to the action invocation.

Returns This function returns a future representing the result of the overall reduction operation.

hpx/collectives/scatter.hpp

Defined in header hpx/collectives/scatter.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Top level HPX namespace.

namespace **collectives**

Top level HPX namespace.

Functions

template<typename **T**>
hpx::*future*<*T*> **scatter_from**(char const *basename, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*(), *root_site_arg* root_site = *root_site_arg*())

Scatter (receive) a set of values to different call sites

This function receives an element of a set of values operating on the given base name.

Parameters

- **basename** – The base name identifying the scatter operation
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **root_site** – The sequence number of the central scatter point (usually the locality id). This value is optional and defaults to 0.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

template<typename **T**>
hpx::*future*<*T*> **scatter_from**(*communicator* comm, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Scatter (receive) a set of values to different call sites

This function receives an element of a set of values operating on the given base name.

Scatter (receive) a set of values to different call sites

This function receives an element of a set of values operating on the given base name.

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **generation** – The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

template<typename **T**>
hpx::*future*<*T*> **scatter_to**(char const *basename, *std*::vector<*T*> &&result, *num_sites_arg* num_sites = *num_sites_arg*(), *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Scatter (send) a part of the value set at the given call site

This function transmits the value given by *result* to a central scatter site (where the corresponding *scatter_from* is executed)

Parameters

- **basename** – The base name identifying the scatter operation
- **result** – The value to transmit to the central scatter point from this call site.
- **num_sites** – The number of participating sites (default: all localities).
- **generation** – The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

template<typename **T**>
hpx::*future*<*T*> **scatter_to**(*communicator* comm, *std*::vector<*T*> &&result, *this_site_arg* this_site = *this_site_arg*(), *generation_arg* generation = *generation_arg*())

Scatter (send) a part of the value set at the given call site

This function transmits the value given by *result* to a central scatter site (where the corresponding *scatter_from* is executed)

Scatter (send) a part of the value set at the given call site

This function transmits the value given by *result* to a central scatter site (where the corresponding *scatter_from* is executed)

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Note: The generation values from corresponding *scatter_to* and *scatter_from* have to match.

Parameters

- **comm** – A communicator object returned from *create_communicator*
- **num_sites** – The number of participating sites (default: all localities).
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.
- **generation** – The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **comm** – A communicator object returned from *create_communicator*
- **num_sites** – The number of participating sites (default: all localities).
- **generation** – The generational counter identifying the sequence number of the all_gather operation performed on the given base name. This is optional and needs to be supplied only if the all_gather operation on the given base name has to be performed more than once. The generation number (if given) must be a positive number greater than zero.
- **this_site** – The sequence number of this invocation (usually the locality id). This value is optional and defaults to whatever hpx::get_locality_id() returns.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

Returns This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

components

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/components/basename_registration.hpp

Defined in header hpx/components/basename_registration.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **Client**>
std::vector<*Client*> **find_all_from_basename**(*std*::string base_name, *std*::size_t num_ids)

Return all registered clients from all localities from the given base name.

This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

Return all registered ids from all localities from the given base name.

This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

Note: The futures embedded in the returned client objects will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Note: The futures will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Template Parameters `Client` – The client type to return

Parameters

- **base_name** – [in] The base name for which to retrieve the registered ids.
- **num_ids** – [in] The number of registered ids to expect.
- **base_name** – [in] The base name for which to retrieve the registered ids.
- **num_ids** – [in] The number of registered ids to expect.

Returns A list of futures representing the ids which were registered using the given base name.

Returns A list of futures representing the ids which were registered using the given base name.

template<typename **Client**>
std::vector<*Client*> **find_from_basename**(*std*::string base_name, *std*::vector<*std*::size_t> const &ids)

Return registered clients from the given base name and sequence numbers.

This function locates the ids which were registered with the given base name and the given sequence numbers. It returns a list of futures representing those ids.

Return registered ids from the given base name and sequence numbers.

This function locates the ids which were registered with the given base name and the given sequence numbers. It returns a list of futures representing those ids.

Note: The futures embedded in the returned client objects will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Note: The futures will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Template Parameters `Client` – The client type to return

Parameters

- **base_name** – [in] The base name for which to retrieve the registered ids.
- **ids** – [in] The sequence numbers of the registered ids.
- **base_name** – [in] The base name for which to retrieve the registered ids.
- **ids** – [in] The sequence numbers of the registered ids.

Returns A list of futures representing the ids which were registered using the given base name and sequence numbers.

Returns A list of futures representing the ids which were registered using the given base name and sequence numbers.

template<typename **Client**>
Client **find_from_basename**(*std*::string base_name, *std*::size_t sequence_nr)

Return registered id from the given base name and sequence number.

This function locates the id which was registered with the given base name and the given sequence number. It returns a future representing those id.

This function locates the id which was registered with the given base name and the given sequence number. It returns a future representing those id.

Note: The future embedded in the returned client object will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Note: The future will become ready even if the event (for instance, binding the name to an id) has already happened in the past. This is important in order to reliably retrieve ids from a name, even if the name was already registered.

Template Parameters `Client` – The client type to return

Parameters

- **base_name** – [in] The base name for which to retrieve the registered ids.
- **sequence_nr** – [in] The sequence number of the registered id.
- **base_name** – [in] The base name for which to retrieve the registered ids.
- **sequence_nr** – [in] The sequence number of the registered id.

Returns A representing the id which was registered using the given base name and sequence numbers.

Returns A representing the id which was registered using the given base name and sequence numbers.

template<typename **Client**, typename **Stub**, typename **Data**>
hpx::*future*<bool> **register_with_basename**(*std*::string base_name, *components*::client_base<*Client*, *Stub*, *Data*> &client, *std*::size_t sequence_nr)

Register the id wrapped in the given client using the given base name.

The function registers the object the given client refers to using the provided base name.

Note: The operation will fail if the given sequence number is not unique.

Template Parameters `Client` – The client type to register

Parameters

- `base_name` – [in] The base name for which to retrieve the registered ids.
- `client` – [in] The client which should be registered using the given base name.
- `sequence_nr` – [in, optional] The sequential number to use for the registration of the id. This number has to be unique system-wide for each registration using the same base name. The default is the current locality identifier. Also, the sequence numbers have to be consecutive starting from zero.

Returns A future representing the result of the registration operation itself.

template<typename **Client**>
Client **unregister_with_basename**(*std*::string base_name, *std*::size_t sequence_nr = ~static_cast<*std*::size_t>(0))

Unregister the given id using the given base name.

Unregister the given base name.

The function unregisters the given ids using the provided base name.

The function unregisters the given ids using the provided base name.

Template Parameters `Client` – The client type to return

Parameters

- `base_name` – [in] The base name for which to retrieve the registered ids.
- `sequence_nr` – [in, optional] The sequential number to use for the un-registration. This number has to be the same as has been used with *register_with_basename* before.
- `base_name` – [in] The base name for which to retrieve the registered ids.
- `sequence_nr` – [in, optional] The sequential number to use for the un-registration. This number has to be the same as has been used with *register_with_basename* before.

Returns A future representing the result of the un-registration operation itself.

Returns A future representing the result of the un-registration operation itself.

hpx/components/basename_registration_fwd.hpp

Defined in header hpx/components/basename_registration_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

hpx::*future*<bool> **register_with_basename**(*std*::string base_name, *hpx*::id_type const &id, *std*::size_t sequence_nr = ~static_cast<*std*::size_t>(0))

Register the given id using the given base name.

The function registers the given ids using the provided base name.

Note: The operation will fail if the given sequence number is not unique.

Parameters

- **base_name** – [in] The base name for which to retrieve the registered ids.
- **id** – [in] The id to register using the given base name.
- **sequence_nr** – [in, optional] The sequential number to use for the registration of the id. This number has to be unique system-wide for each registration using the same base name. The default is the current locality identifier. Also, the sequence numbers have to be consecutive starting from zero.

Returns A future representing the result of the registration operation itself.

bool **register_with_basename**(*hpx*::*launch*::sync_policy, *std*::string base_name, *hpx*::id_type const &id, *std*::size_t sequence_nr = ~static_cast<*std*::size_t>(0), *error_code* &ec = *throws*)

hpx::*future*<bool> **register_with_basename**(*std*::string base_name, *hpx*::*future*<*hpx*::id_type> f, *std*::size_t sequence_nr = ~static_cast<*std*::size_t>(0))

Register the id wrapped in the given future using the given base name.

The function registers the object the given future refers to using the provided base name.

Note: The operation will fail if the given sequence number is not unique.

Parameters

- **base_name** – [in] The base name for which to retrieve the registered ids.
- **f** – [in] The future which should be registered using the given base name.
- **sequence_nr** – [in, optional] The sequential number to use for the registration of the id. This number has to be unique system-wide for each registration using the same base name. The default is the current locality identifier. Also, the sequence numbers have to be consecutive starting from zero.

Returns A future representing the result of the registration operation itself.

hpx/components/get_ptr.hpp

Defined in header hpx/components/get_ptr.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename **Component**>
hpx::*future*<*std*::shared_ptr<*Component*>> **get_ptr**(*hpx*::id_type const &id)

Returns a future referring to the pointer to the underlying memory of a component.

The function hpx::get_ptr can be used to extract a future referring to the pointer to the underlying memory of a given component.

Note: This function will successfully return the requested result only if the given component is currently located on the calling locality. Otherwise, the function will raise an error.

Note: The component instance the returned pointer refers to can not be migrated as long as there is at least one copy of the returned shared_ptr alive.

Parameters **id** – [in] The global id of the component for which the pointer to the underlying memory should be retrieved.

Template Parameters **Component** – The type of the server side component.

Returns This function returns a future representing the pointer to the underlying memory for the component instance with the given *id*.

template<typename **Derived**, typename **Stub**, typename **Data**>
hpx::*future*<*std*::shared_ptr<typename *components*::client_base<*Derived*, *Stub*, *Data*>::server_component_type>> **get_ptr**(*com*
Stu
Dat
con
&c)

Returns a future referring to the pointer to the underlying memory of a component.

The function hpx::get_ptr can be used to extract a future referring to the pointer to the underlying memory of a given component.

Note: This function will successfully return the requested result only if the given component is currently located on the calling locality. Otherwise, the function will raise an error.

Note: The component instance the returned pointer refers to can not be migrated as long as there is at least one copy of the returned shared_ptr alive.

Parameters **c** – [in] A client side representation of the component for which the pointer to the underlying memory should be retrieved.

Returns This function returns a future representing the pointer to the underlying memory for the component instance with the given *id*.

template<typename **Component**>
std::shared_ptr<*Component*> **get_ptr**(*launch*::sync_policy p, *hpx*::id_type const &id, *error_code* &ec = *throws*)

Returns the pointer to the underlying memory of a component.

The function hpx::get_ptr_sync can be used to extract the pointer to the underlying memory of a given component.

Note: This function will successfully return the requested result only if the given component is currently located on the requesting locality. Otherwise, the function will raise and error.

Note: The component instance the returned pointer refers to can not be migrated as long as there is at least one copy of the returned shared_ptr alive.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise, it throws an instance of hpx::exception.

Parameters

- **p** – [in] The parameter *p* represents a placeholder type to turn make the call synchronous.
- **id** – [in] The global id of the component for which the pointer to the underlying memory should be retrieved.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Template Parameters **Component** – The only template parameter has to be the type of the server side component.

Returns This function returns the pointer to the underlying memory for the component instance with the given *id*.

template<typename **Derived**, typename **Stub**, typename **Data**>

std::shared_ptr<typename *components*::client_base<*Derived*, *Stub*, *Data*>::server_component_type> **get_ptr**(*launch*::sync_pol p, *components*::client_base<*Derived*, *Stub*, *Data*> const &c, *error_code* &ec = *throws*)

Returns the pointer to the underlying memory of a component.

The function hpx::get_ptr_sync can be used to extract the pointer to the underlying memory of a given component.

Note: This function will successfully return the requested result only if the given component is currently located on the requesting locality. Otherwise, the function will raise and error.

Note: The component instance the returned pointer refers to can not be migrated as long as there is at least one copy of the returned shared_ptr alive.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise, it throws an instance of hpx::exception.

Parameters

- **p** – [in] The parameter *p* represents a placeholder type to turn make the call synchronous.
- **c** – [in] A client side representation of the component for which the pointer to the underlying memory should be retrieved.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function returns the pointer to the underlying memory for the component instance with the given *id*.

components_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/components_base/agas_interface.hpp

Defined in header hpx/components_base/agas_interface.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **agas**

Functions

bool **is_console**()

bool **register_name**(*launch*::sync_policy, *std*::string const &name, *naming*::gid_type const &gid, *error_code* &ec = *throws*)

bool **register_name**(*launch*::sync_policy, *std*::string const &name, *hpx*::id_type const &id, *error_code* &ec = *throws*)

hpx::*future*<bool> **register_name**(*std*::string const &name, *hpx*::id_type const &id)

hpx::id_type **unregister_name**(*launch*::sync_policy, *std*::string const &name, *error_code* &ec = *throws*)

hpx::*future*<*hpx*::id_type> **unregister_name**(*std*::string const &name)

hpx::id_type **resolve_name**(*launch*::sync_policy, *std*::string const &name, *error_code* &ec = *throws*)

hpx::*future*<*hpx*::id_type> **resolve_name**(*std*::string const &name)

hpx::*future*<*std*::uint32_t> **get_num_localities**(*naming*::component_type type = *naming*::component_invalid)

std::uint32_t **get_num_localities**(*launch*::sync_policy, *naming*::component_type type, *error_code* &ec = *throws*)

inline *std*::uint32_t **get_num_localities**(*launch*::sync_policy, *error_code* &ec = *throws*)

std::string **get_component_type_name**(*naming*::component_type type, *error_code* &ec = *throws*)

hpx::*future*<*std*::vector<*std*::uint32_t>> **get_num_threads**()

std::vector<*std*::uint32_t> **get_num_threads**(*launch*::sync_policy, *error_code* &ec = *throws*)

hpx::*future*<*std*::uint32_t> **get_num_overall_threads**()

std::uint32_t **get_num_overall_threads**(*launch*::sync_policy, *error_code* &ec = *throws*)

std::uint32_t **get_locality_id**(*error_code* &ec = *throws*)

inline *hpx*::*naming*::gid_type **get_locality**()

std::vector<*std*::uint32_t> **get_all_locality_ids**(*naming*::component_type type, *error_code* &ec = *throws*) inline std::vector<std::uint32_t> **get_all_locality_ids**(error_code &ec = throws)

bool **is_local_address_cached**(naming::gid_type const &gid, error_code &ec = throws)

bool **is_local_address_cached**(naming::gid_type const &gid, naming::address &addr, error_code &ec = throws)

bool **is_local_address_cached**(naming::gid_type const &gid, naming::address &addr, std::pair<bool, components::pinned_ptr> &r, hpx::move_only_function<std::pair<bool, components::pinned_ptr>(naming::address const&)> &&f, error_code &ec = throws)

inline bool **is_local_address_cached**(hpx::id_type const &id, error_code &ec = throws)

inline bool **is_local_address_cached**(hpx::id_type const &id, naming::address &addr, error_code &ec = throws)

inline bool **is_local_address_cached**(hpx::id_type const &id, naming::address &addr, std::pair<bool, components::pinned_ptr> &r, hpx::move_only_function<std::pair<bool, components::pinned_ptr>(naming::address const&)> &&f, error_code &ec = throws)

void **update_cache_entry**(naming::gid_type const &gid, naming::address const &addr, std::uint64_t count = 0, std::uint64_t offset = 0, error_code &ec = throws)

bool **is_local_lva_encoded_address**(naming::gid_type const &gid)

inline bool **is_local_lva_encoded_address**(hpx::id_type const &id)

hpx::future_or_value<naming::address> **resolve_async**(hpx::id_type const &id)

hpx::future<naming::address> **resolve**(hpx::id_type const &id)

naming::address **resolve**(launch::sync_policy, hpx::id_type const &id, error_code &ec = throws)

bool **resolve_local**(naming::gid_type const &gid, naming::address &addr, error_code &ec = throws)

bool **resolve_cached**(naming::gid_type const &gid, naming::address &addr)

hpx::future<bool> **bind**(naming::gid_type const &gid, naming::address const &addr, std::uint32_t locality_id)

bool **bind**(launch::sync_policy, naming::gid_type const &gid, naming::address const &addr, std::uint32_t locality_id, error_code &ec = throws)

hpx::future<bool> **bind**(naming::gid_type const &gid, naming::address const &addr, naming::gid_type const &locality_)

bool **bind**(launch::sync_policy, naming::gid_type const &gid, naming::address const &addr, naming::gid_type const &locality_, error_code &ec = throws)

hpx::future<naming::address> **unbind**(naming::gid_type const &gid, std::uint64_t count = 1)

naming::address **unbind**(launch::sync_policy, naming::gid_type const &gid, std::uint64_t count = 1, error_code &ec = throws) bool **bind_gid_local**(*naming*::gid_type const &gid, *naming*::address const &addr, *error_code* &ec = *throws*)

void **unbind_gid_local**(*naming*::gid_type const &gid, *error_code* &ec = *throws*)

bool **bind_range_local**(*naming*::gid_type const &gid, *std*::size_t count, *naming*::address const &addr, *std*::size_t offset, *error_code* &ec = *throws*)

void **unbind_range_local**(*naming*::gid_type const &gid, *std*::size_t count, *error_code* &ec = *throws*)

void **garbage_collect_non_blocking**(*error_code* &ec = *throws*)

void **garbage_collect**(*error_code* &ec = *throws*)

void **garbage_collect_non_blocking**(*hpx*::id_type const &id, *error_code* &ec = *throws*)

Invoke an asynchronous garbage collection step on the given target locality.

void **garbage_collect**(*hpx*::id_type const &id, *error_code* &ec = *throws*)

Invoke a synchronous garbage collection step on the given target locality.

hpx::id_type **get_console_locality**(*error_code* &ec = *throws*)

Return an id_type referring to the console locality.

naming::gid_type **get_next_id**(*std*::size_t count, *error_code* &ec = *throws*)

void **decref**(*naming*::gid_type const &id, *std*::int64_t credits, *error_code* &ec = *throws*)

hpx::future_or_value<*std*::int64_t> **incref**(*naming*::gid_type const &gid, *std*::int64_t credits, *hpx*::id_type const &keep_alive = *hpx*::invalid_id)

std::int64_t **incref**(*launch*::sync_policy, *naming*::gid_type const &gid, *std*::int64_t credits = 1, *hpx*::id_type const &keep_alive = *hpx*::invalid_id, *error_code* &ec = *throws*)

std::int64_t **replenish_credits**(*naming*::gid_type &gid)

hpx::future_or_value<id_type> **get_colocation_id**(*hpx*::id_type const &id)

hpx::id_type **get_colocation_id**(*launch*::sync_policy, *hpx*::id_type const &id, *error_code* &ec = *throws*)

hpx::*future*<*hpx*::id_type> **on_symbol_namespace_event**(*std*::string const &name, bool call_for_past_events)

hpx::*future*<*std*::pair<*hpx*::id_type, *naming*::address>> **begin_migration**(*hpx*::id_type const &id)

bool **end_migration**(*hpx*::id_type const &id)

hpx::*future*<void> **mark_as_migrated**(*naming*::gid_type const &gid, *hpx*::*move_only_function*<*std*::pair<bool, *hpx*::*future*<void>>()> &&f, bool expect_to_be_marked_as_migrating)

std::pair<bool, *components*::pinned_ptr> **was_object_migrated**(*naming*::gid_type const &gid, *hpx*::*move_only_function*<*components*::pinned_ptr()> &&f)

void **unmark_as_migrated**(*naming*::gid_type const &gid, *hpx*::*move_only_function*<void()> &&f)

hpx::*future*<*std*::map<*std*::string, *hpx*::id_type>> **find_symbols**(*std*::string const &pattern = "*")

std::map<*std*::string, *hpx*::id_type> **find_symbols**(*hpx*::*launch*::sync_policy, *std*::string const &pattern = "*")

naming::component_type **register_factory**(*std*::uint32_t prefix, *std*::string const &name, *error_code* &ec = *throws*)

naming::component_type **get_component_id**(*std*::string const &name, *error_code* &ec = *throws*)

void **destroy_component**(*naming*::gid_type const &gid, *naming*::address const &addr)

naming::address_type **get_primary_ns_lva**()

naming::address_type **get_symbol_ns_lva**()

naming::address_type **get_runtime_support_lva**()

struct agas_interface_functions &**agas_init**()

hpx/components_base/component_commandline.hpp

Defined in header hpx/components_base/component_commandline.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_DEFINE_COMPONENT_COMMANDLINE_OPTIONS(add_options_function)

HPX_REGISTER_COMMANDLINE_MODULE(add_options_function)

HPX_REGISTER_COMMANDLINE_MODULE_DYNAMIC(add_options_function)

namespace **hpx**

namespace **components**

struct **component_commandline** : public *component_commandline_base*

#include <component_commandline.hpp> The component_startup_shutdown provides a minimal implementation of a component's startup/shutdown function provider.

Public Functions

inline *hpx*::program_options::options_description **add_commandline_options**() override

Return any additional command line options valid for this component.

Note: This function will be executed by the runtime system during system startup.

Returns The module is expected to fill a options_description object with any additional command line options this component will handle.

namespace **commandline_options_provider**

Functions

hpx::program_options::options_description **add_commandline_options**()

hpx/components_base/component_startup_shutdown.hpp

Defined in header hpx/components_base/component_startup_shutdown.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_DEFINE_COMPONENT_STARTUP_SHUTDOWN(startup_, shutdown_)

HPX_REGISTER_STARTUP_SHUTDOWN_MODULE_(startup, shutdown)

HPX_REGISTER_STARTUP_SHUTDOWN_MODULE(startup, shutdown)

HPX_REGISTER_STARTUP_SHUTDOWN_MODULE_DYNAMIC(startup, shutdown)

HPX_REGISTER_STARTUP_MODULE(startup)

HPX_REGISTER_STARTUP_MODULE_DYNAMIC(startup)

HPX_REGISTER_SHUTDOWN_MODULE(shutdown)

HPX_REGISTER_SHUTDOWN_MODULE_DYNAMIC(shutdown)

namespace **hpx**

namespace **components**

template<bool (***Startup**)(*startup_function_type*&, bool&), bool (***Shutdown**)(*shutdown_function_type*&, bool&)>
struct **component_startup_shutdown** : public *component_startup_shutdown_base*

#include <component_startup_shutdown.hpp> The component_startup_shutdown class provides a minimal implementation of a component's startup/shutdown function provider.

Public Functions

inline bool **get_startup_function**(*startup_function_type* &startup, bool &pre_startup) override

Return any startup function for this component.

Parameters

- **startup** – [in, out] The module is expected to fill this function object with a reference to a startup function. This function will be executed by the runtime system during system startup.
- **pre_startup** –

Returns Returns *true* if the parameter *startup* has been successfully initialized with the startup function.

inline bool **get_shutdown_function**(*shutdown_function_type* &shutdown, bool &pre_shutdown) override

Return any startup function for this component.

Parameters

- **shutdown** – [in, out] The module is expected to fill this function object with a reference to a startup function. This function will be executed by the runtime system during system startup.
- **pre_shutdown** –

Returns Returns *true* if the parameter *shutdown* has been successfully initialized with the shutdown function.

hpx/components_base/component_type.hpp

Defined in header hpx/components_base/component_type.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_COMPONENT_ENUM_TYPE_ENUM_DEPRECATION_MSG

HPX_FACTORY_STATE_ENUM_DEPRECATION_MSG

HPX_DEFINE_GET_COMPONENT_TYPE(component)

HPX_DEFINE_GET_COMPONENT_TYPE_TEMPLATE(template_, component)

HPX_DEFINE_GET_COMPONENT_TYPE_STATIC(component, type)

HPX_DEFINE_COMPONENT_NAME(...)

HPX_DEFINE_COMPONENT_NAME_(...)

HPX_DEFINE_COMPONENT_NAME_2(Component, name)

HPX_DEFINE_COMPONENT_NAME_3(Component, name, base_name)

namespace **hpx**

namespace **components**

Typedefs

using **component_deleter_type** = void (*)(*hpx*::*naming*::gid_type const&, *hpx*::*naming*::address const&)

Enums

enum class **component_enum_type** : *naming*::component_type

Values:

enumerator **invalid**

enumerator **runtime_support**

enumerator **plain_function**

enumerator **base_lco**

enumerator **base_lco_with_value_unmanaged**

enumerator **base_lco_with_value**

enumerator **latch**

enumerator **barrier**

enumerator **promise**

enumerator **agas_locality_namespace**

enumerator **agas_primary_namespace**

enumerator **agas_component_namespace**

enumerator **agas_symbol_namespace**

enumerator **last**

enumerator **first_dynamic**

enum class **factory_state** : *std*::uint8_t

Values:

enumerator **enabled**

enumerator **disabled**

enumerator **check**

Functions

constexpr *naming*::component_type **to_int**(*component_enum_type* t) noexcept

constexpr int **to_int**(*factory_state* t) noexcept

bool &**enabled**(component_type type)

util::atomic_count &**instance_count**(component_type type)

component_deleter_type &**deleter**(component_type type)

bool **enumerate_instance_counts**(*hpx*::*move_only_function*<bool(component_type)> const &f)

std::string **get_component_type_name**(component_type type)

Return the string representation for a given component type id.

constexpr component_type **get_base_type**(component_type t) noexcept

The lower short word of the component type is the type of the component exposing the actions.

constexpr component_type **get_derived_type**(component_type t) noexcept

The upper short word of the component is the actual component type.

constexpr component_type **derived_component_type**(component_type derived, component_type base) noexcept

A component derived from a base component exposing the actions needs to have a specially formatted component type.

constexpr bool **types_are_compatible**(component_type lhs, component_type rhs) noexcept

Verify the two given component types are matching (compatible)

template<typename **Component**, typename **Enable** = void>
char const ***get_component_name**() noexcept

template<typename **Component**, typename **Enable** = void>
char const ***get_component_base_name**() noexcept

template<typename **Component**>
component_type **get_component_type**() noexcept

template<typename **Component**>
void **set_component_type**(component_type type)

Variables

constexpr *component_enum_type* **component_invalid** = *component_enum_type*::*invalid*

constexpr *component_enum_type* **component_runtime_support** = *component_enum_type*::*runtime_support*

constexpr *component_enum_type* **component_plain_function** = *component_enum_type*::*plain_function*

constexpr *component_enum_type* **component_base_lco** = *component_enum_type*::*base_lco* constexpr *component_enum_type* **component_base_lco_with_value_unmanaged** = *component_enum_type*::*base_lco_with_value_unmanaged*

constexpr *component_enum_type* **component_base_lco_with_value** = *component_enum_type*::*base_lco_with_value*

constexpr *component_enum_type* **component_latch** = *component_enum_type*::*latch*

constexpr *component_enum_type* **component_barrier** = *component_enum_type*::*barrier*

constexpr *component_enum_type* **component_promise** = *component_enum_type*::*promise*

constexpr *component_enum_type* **component_agas_locality_namespace** = *component_enum_type*::*agas_locality_namespace*

constexpr *component_enum_type* **component_agas_primary_namespace** = *component_enum_type*::*agas_primary_namespace*

constexpr *component_enum_type* **component_agas_component_namespace** = *component_enum_type*::*agas_component_namespace*

constexpr *component_enum_type* **component_agas_symbol_namespace** = *component_enum_type*::*agas_symbol_namespace*

constexpr *component_enum_type* **component_last** = *component_enum_type*::*last*

constexpr *component_enum_type* **component_first_dynamic** = *component_enum_type*::*first_dynamic*

constexpr *factory_state* **factory_enabled** = *factory_state*::*enabled*

constexpr *factory_state* **factory_disabled** = *factory_state*::*disabled*

constexpr *factory_state* **factory_check** = *factory_state*::*check*

hpx/components_base/get_lva.hpp

Defined in header hpx/components_base/get_lva.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

template<typename **Component**, typename **Enable** = void>

struct **get_lva**

#include <get_lva.hpp> The get_lva template is a helper structure allowing to convert a local virtual address as stored in a local address (returned from the function *resolver_client::resolve*) to the address of the component implementing the action.

The default implementation uses the template argument *Component* to deduce the type wrapping the component implementing the action. This is used to get the needed address.

Template Parameters **Component** – This is the type of the component implementing the action to execute.

Public Static Functions

static inline constexpr *Component* ***call**(*naming*::address_type lva) noexcept

hpx/components_base/server/fixed_component_base.hpp

Defined in header hpx/components_base/server/fixed_component_base.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

template<typename **Component**>

class **fixed_component** : public *Component*

Public Types

using **type_holder** = *Component*

using **component_type** = *fixed_component*<*Component*>

using **derived_type** = *component_type*

using **heap_type** = detail::fixed_heap

Public Static Functions

static inline *Component* ***create***(*std*::size_t) noexcept

The function *create* is used for allocation and initialization of instances of the derived components.

static inline void **destroy**(*Component**, *std*::size_t = 1) noexcept

The function *destroy* is used for destruction and de-allocation of instances of the derived components.

template<typename **Component**>

class **fixed_component_base** : public fixed_component_tag

Public Types

using **wrapped_type** = *this_component_type*

using **base_type_holder** = *this_component_type*

using **wrapping_type** = *fixed_component*<*this_component_type*>

Public Functions

inline constexpr **fixed_component_base**(*std*::uint64_t msb, *std*::uint64_t lsb) noexcept

~fixed_component_base() = default

inline void **finalize**() const

finalize() will be called just before the instance gets destructed

inline *naming*::gid_type **get_base_gid**(*naming*::gid_type const &assign_gid = *naming*::invalid_gid) const

inline *hpx*::id_type **get_id**() const

inline *hpx*::id_type **get_unmanaged_id**() const

inline void **set_locality_id**(*std*::uint32_t locality_id, *error_code* &ec = *throws*)

Public Static Functions

static inline void **mark_as_migrated**() noexcept

static inline void **on_migrated**() noexcept

Private Types

using **this_component_type** = *std*::conditional_t<*std*::is_void_v<*Component*>, *fixed_component_base*, *Component*>

Private Functions

inline constexpr *Component* &**derived**() noexcept

inline constexpr *Component* const &**derived**() const noexcept

Private Members

mutable *naming*::gid_type **gid_**

std::uint64_t **msb_**

std::uint64_t **lsb_**

hpx/components_base/server/managed_component_base.hpp

Defined in header hpx/components_base/server/managed_component_base.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

template<>

struct `hpx::components::detail_adl_barrier::init`<*traits*::construct_with_back_ptr>

Public Static Functions

template<typename **Component**, typename **Managed**>
static inline constexpr void **call**(*Component**, *Managed**) noexcept

template<typename **Component**, typename **Managed**, typename ...**Ts**>
static inline void **call_new**(*Component* *&component, *Managed* *this_, *Ts*&&... vs)

template<>

struct `hpx::components::detail_adl_barrier::init`<*traits*::construct_without_back_ptr>

Public Static Functions

template<typename **Component**, typename **Managed**>
static inline void **call**(*Component* *component, *Managed* *this_)

template<typename **Component**, typename **Managed**, typename ...**Ts**>
static inline void **call_new**(*Component* *&component, *Managed* *this_, *Ts*&&... vs)

template<>

struct
hpx::*components*::*detail_adl_barrier*::**destroy_backptr**<*traits*::managed_object_is_lifetime_controlled>

Public Static Functions

template<typename **BackPtr**>
static inline void **call**(*BackPtr* *back_ptr)

template<>

struct
hpx::*components*::*detail_adl_barrier*::**destroy_backptr**<*traits*::managed_object_controls_lifetime>

Public Static Functions

template<typename **BackPtr**>
static inline constexpr void **call**(*BackPtr**) noexcept

template<>

struct
hpx::*components*::*detail_adl_barrier*::**manage_lifetime**<*traits*::managed_object_is_lifetime_controlled>

Public Static Functions

template<typename **Component**>
static inline constexpr void **call**(*Component**) noexcept

template<typename **Component**>
static inline void **addref**(*Component* *component) noexcept

template<typename **Component**>
static inline void **release**(*Component* *component) noexcept

template<>

struct
hpx::*components*::*detail_adl_barrier*::**manage_lifetime**<*traits*::managed_object_controls_lifetime>

Public Static Functions

template<typename **Component**>
static inline void **call**(*Component* *component) noexcept(noexcept(*component*->finalize()))

template<typename **Component**>
static inline constexpr void **addref**(*Component**) noexcept

template<typename **Component**>
static inline constexpr void **release**(*Component**) noexcept

namespace **hpx**

namespace **components**

Functions

template<typename **Component**, typename **Derived**>
void **intrusive_ptr_add_ref**(*managed_component*<*Component*, *Derived*> *p) noexcept

template<typename **Component**, typename **Derived**>
void **intrusive_ptr_release**(*managed_component*<*Component*, *Derived*> *p) noexcept

template<typename **Component**, typename **Derived**>

class **managed_component**

#include <managed_component_base.hpp> The *managed_component* template is used as an indirection layer for components allowing to gracefully handle the access to non-existing components.

Additionally, it provides memory management capabilities for the wrapping instances, and it integrates the memory management with the AGAS service. Every instance of a *managed_component* gets assigned a global id. The provided memory management allocates the *managed_component* instances from a special heap, ensuring fast allocation and avoids a full network round trip to the AGAS service for each of the allocated instances.

Template Parameters

- **Component** – Component type
- **Derived** – Most derived component type

Public Types

using **derived_type** = *std*::conditional_t<*std*::is_void_v<*Derived*>, *managed_component*, *Derived*>

using **wrapped_type** = *Component*

using **type_holder** = *Component*

using **base_type_holder** = typename *Component*::base_type_holder using **heap_type** = detail::wrapper_heap_list<detail::fixed_wrapper_heap<*derived_type*>>

using **value_type** = *derived_type*

Public Functions

managed_component(*managed_component* const&) = delete

managed_component(*managed_component*&&) = delete

managed_component &**operator=**(*managed_component* const&) = delete

managed_component &**operator=**(*managed_component*&&) = delete

inline explicit **managed_component**(*Component* *comp)

Construct a *managed_component* instance holding a wrapped instance. This constructor takes ownership of the passed pointer.

Parameters comp – [in] The pointer to the wrapped instance. The *managed_component* takes ownership of this pointer.

inline **managed_component**()

template<typename **T**, typename ...**Ts**, typename **Enable** = *std*::enable_if_t<!*std*::is_same_v<*std*::decay_t<*T*>, *managed_component*>>>
inline explicit **managed_component**(*T* &&t, *Ts*&&... ts)

inline **~managed_component**()

inline constexpr *Component* ***get**() noexcept

Return a pointer to the wrapped instance.

Note: Caller must check validity of returned pointer

inline constexpr *Component* const ***get**() const noexcept

inline *Component* ***get_checked**()

inline *Component* const ***get_checked**() const

inline *Component* ***operator->**()

inline *Component* const ***operator->**() const

inline *Component* &**operator***()

inline *Component* const &**operator***() const

inline *hpx*::id_type **get_unmanaged_id**() const

Return the global id of this *future* instance.

inline *naming*::gid_type **get_base_gid**(*naming*::gid_type const &assign_gid = *naming*::invalid_gid) const

Public Static Functions

static inline constexpr void **finalize**() noexcept

Protected Attributes

Component ***component_** = nullptr

Friends

template<typename **C**, typename **D**>
friend void **intrusive_ptr_add_ref**(*managed_component*<*C*, *D*> *p) noexcept

template<typename **C**, typename **D**>
friend void **intrusive_ptr_release**(*managed_component*<*C*, *D*> *p) noexcept

template<typename **Component**, typename **Wrapper**, typename **CtorPolicy**, typename **DtorPolicy**>

class **managed_component_base** : public *hpx*::*components*::detail::base_managed_component

Public Types

using **this_component_type** = *std*::conditional_t<*std*::is_void_v<*Component*>, *managed_component_base*, *Component*>

using **wrapped_type** = *this_component_type*

using **has_managed_component_base** = void

using **ctor_policy** = *CtorPolicy*

using **dtor_policy** = *DtorPolicy*

using **wrapping_type** = *managed_component*<*Component*, *Wrapper*>

using **base_type_holder** = *Component*

Public Functions

managed_component_base(*managed_component_base* const&) = delete

managed_component_base(*managed_component_base*&&) = delete

managed_component_base &**operator=**(*managed_component_base* const&) = delete

managed_component_base &**operator=**(*managed_component_base*&&) = delete constexpr **managed_component_base**() noexcept = default

inline explicit **managed_component_base**(*managed_component*<*Component*, *Wrapper*> *back_ptr) noexcept

inline **~managed_component_base**()

hpx::id_type **get_unmanaged_id**() const

hpx::id_type **get_id**() const

Protected Functions

naming::gid_type **get_base_gid**() const

inline void **set_back_ptr**(*components*::*managed_component*<*Component*, *Wrapper*> *bp) noexcept

Private Members

managed_component<*Component*, *Wrapper*> ***back_ptr_** = nullptr

Friends

friend struct detail_adl_barrier::init

namespace **detail_adl_barrier**

template<typename **DtorTag**>

struct **destroy_backptr**

template<> managed_object_controls_lifetime >

Public Static Functions

template<typename **BackPtr**>
static inline constexpr void **call**(*BackPtr**) noexcept

template<> managed_object_is_lifetime_controlled >

Public Static Functions

template<typename **BackPtr**>
static inline void **call**(*BackPtr* *back_ptr)

template<typename **BackPtrTag**>

struct **init**

template<> construct_with_back_ptr >

Public Static Functions

template<typename **Component**, typename **Managed**>
static inline constexpr void **call**(*Component**, *Managed**) noexcept

template<typename **Component**, typename **Managed**, typename ...**Ts**>
static inline void **call_new**(*Component* *&component, *Managed* *this_, *Ts*&&... vs)

template<> construct_without_back_ptr >

Public Static Functions

template<typename **Component**, typename **Managed**>
static inline void **call**(*Component* *component, *Managed* *this_)

template<typename **Component**, typename **Managed**, typename ...**Ts**>
static inline void **call_new**(*Component* *&component, *Managed* *this_, *Ts*&&... vs)

template<typename **DtorTag**>

struct **manage_lifetime**

template<> managed_object_controls_lifetime >

Public Static Functions

template<typename **Component**>
static inline void **call**(*Component* *component) noexcept(noexcept(*component*->finalize()))

template<typename **Component**>
static inline constexpr void **addref**(*Component**) noexcept

template<typename **Component**>
static inline constexpr void **release**(*Component**) noexcept

template<> managed_object_is_lifetime_controlled >

Public Static Functions

template<typename **Component**>
static inline constexpr void **call**(*Component**) noexcept

template<typename **Component**>
static inline void **addref**(*Component* *component) noexcept

template<typename **Component**>
static inline void **release**(*Component* *component) noexcept

hpx/components_base/server/migration_support.hpp

Defined in header hpx/components_base/server/migration_support.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

template<typename **BaseComponent**, typename **Mutex** = *hpx*::*spinlock*>

struct **migration_support** : public *BaseComponent*

#include <migration_support.hpp> This hook has to be inserted into the derivation chain of any component for it to support migration.

Public Types

using **decorates_action** = void

Public Functions

inline **migration_support**()

template<typename **T**, typename ...**Ts**, typename =
std::enable_if_t<!*std*::is_same_v<*std*::decay_t<*T*>, *migration_support*>>>
inline explicit **migration_support**(*T* &&t, *Ts*&&... ts)

migration_support(*migration_support* const&) = default

migration_support(*migration_support*&&) = default

migration_support &**operator=**(*migration_support* const&) = default

migration_support &**operator=**(*migration_support*&&) = default

~migration_support() = default

inline *naming*::gid_type **get_base_gid**(*naming*::gid_type const &assign_gid = *naming*::invalid_gid) const inline void **pin**() noexcept

inline bool **unpin**()

inline *std*::uint32_t **pin_count**() const noexcept

inline void **mark_as_migrated**()

inline *hpx*::*future*<void> **mark_as_migrated**(*hpx*::id_type const &to_migrate)

inline void **unmark_as_migrated**(*hpx*::id_type const &to_migrate)

Public Static Functions

static inline constexpr bool **supports_migration**() noexcept

static inline constexpr void **on_migrated**() noexcept

template<typename **F**>
static inline *threads*::thread_function_type **decorate_action**(*naming*::address_type lva, *F* &&f)

static inline *std*::pair<bool, *components*::pinned_ptr> **was_object_migrated**(*hpx*::*naming*::gid_type const &id, *naming*::address_type lva)

Protected Functions

inline *threads*::thread_result_type **thread_function**(*threads*::thread_function_type &&f, *components*::pinned_ptr, *threads*::*thread_restart_state* state)

Private Types

using **base_type** = *BaseComponent*

using **this_component_type** = typename *base_type*::this_component_type

Private Members

hpx::intrusive_ptr<detail::migration_support_data<*Mutex*>> **data_**

hpx::*promise*<void> **trigger_migration_**

bool **started_migration_** = false

bool **was_marked_for_migration_** = false

compute

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/compute/host/target_distribution_policy.hpp

Defined in header hpx/compute/host/target_distribution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

distribution_policies

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/distribution_policies/binpacking_distribution_policy.hpp

Defined in header hpx/distribution_policies/binpacking_distribution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

Variables

constexpr char const *const **default_binpacking_counter_name** = "/runtime{locality/total}/count/component@"

static const *binpacking_distribution_policy* **binpacked** = {}

A predefined instance of the binpacking *distribution_policy*. It will represent the local locality and will place all items to create here.

struct **binpacking_distribution_policy**

#include <binpacking_distribution_policy.hpp> This class specifies the parameters for a binpacking distribution policy to use for creating a given number of items on a given set of localities. The binpacking policy will distribute the new objects in a way such that each of the localities will equalize the number of overall objects of this type based on a given criteria (by default this criteria is the overall number of objects of this type).

Public Functions

inline **binpacking_distribution_policy**()

Default-construct a new instance of a binpacking_distribution_policy. This policy will represent one locality (the local locality).

inline *binpacking_distribution_policy* **operator()**(*std*::vector<id_type> const &locs, char const *perf_counter_name = *default_binpacking_counter_name*) const

Create a new *default_distribution* policy representing the given set of localities.

Parameters
- **locs** – [in] The list of localities the new instance should represent
- **perf_counter_name** – [in] The name of the performance counter which should be used as the distribution criteria (by default the overall number of existing instances of the given component type will be used).

inline *binpacking_distribution_policy* **operator()**(*std*::vector<id_type> &&locs, char const *perf_counter_name = *default_binpacking_counter_name*) const

Create a new *default_distribution* policy representing the given set of localities.

Parameters
- **locs** – [in] The list of localities the new instance should represent
- **perf_counter_name** – [in] The name of the performance counter which should be used as the distribution criteria (by default the overall number of existing instances of the given component type will be used).

inline *binpacking_distribution_policy* **operator()**(id_type const &loc, char const *perf_counter_name = *default_binpacking_counter_name*) const

Create a new *default_distribution* policy representing the given locality

Parameters
- **loc** – [in] The locality the new instance should represent
- **perf_counter_name** – [in] The name of the performance counter that should be used as the distribution criteria (by default the overall number of existing instances of the given component type will be used).

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*hpx*::id_type> **create**(*Ts*&&... vs) const

Create one object on one of the localities associated by this policy instance

Parameters **vs** – [in] The arguments which will be forwarded to the constructor of the new object.

Returns A future holding the global address which represents the newly created object

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*std*::vector<bulk_locality_result>> **bulk_create**(*std*::size_t count, *Ts*&&... vs) const

Create multiple objects on the localities associated by this policy instance

Parameters
- **count** – [in] The number of objects to create
- **vs** – [in] The arguments which will be forwarded to the constructors of the new objects.

Returns A future holding the list of global addresses which represent the newly created objects

inline *std*::string const &**get_counter_name**() const

Returns the name of the performance counter associated with this policy instance.

inline *std*::size_t **get_num_localities**() const

Returns the number of associated localities for this distribution policy

Note: This function is part of the creation policy implemented by this class

hpx/distribution_policies/colocating_distribution_policy.hpp

Defined in header hpx/distribution_policies/colocating_distribution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

Variables

static const *colocating_distribution_policy* **colocated** = { }

A predefined instance of the co-locating *distribution_policy*. It will represent the local locality and will place all items to create here.

struct **colocating_distribution_policy**

#include <colocating_distribution_policy.hpp> This class specifies the parameters for a distribution policy to use for creating a given number of items on the locality where a given object is currently placed.

Public Functions

constexpr **colocating_distribution_policy**() = default

Default-construct a new instance of a colocating_distribution_policy. This policy will represent the local locality.

inline *colocating_distribution_policy* **operator()**(id_type const &id) const

Create a new colocating_distribution_policy representing the locality where the given object is current located

Parameters **id** – [in] The global address of the object with which the new instances should be colocated on

template<typename **Client**, typename **Stub**, typename **Data**>
inline *colocating_distribution_policy* **operator()**(client_base<*Client*, *Stub*, *Data*> const &client) const

Create a new colocating_distribution_policy representing the locality where the given object is current located

Parameters **client** – [in] The client side representation of the object with which the new instances should be colocated on

template<typename **Component**, typename ...**Ts**>

inline *hpx*::*future*<*hpx*::id_type> **create**(*Ts*&&... vs) const

Create one object on the locality of the object this distribution policy instance is associated with

Note: This function is part of the placement policy implemented by this class

Parameters **vs** – [in] The arguments which will be forwarded to the constructor of the new object.

Returns A future holding the global address which represents the newly created object

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*std*::vector<bulk_locality_result>> **bulk_create**(*std*::size_t count, *Ts*&&... vs) const

Create multiple objects colocated with the object represented by this policy instance

Note: This function is part of the placement policy implemented by this class

Parameters

- **count** – [in] The number of objects to create
- **vs** – [in] The arguments which will be forwarded to the constructors of the new objects.

Returns A future holding the list of global addresses which represent the newly created objects

template<typename **Action**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async**(*launch* policy, *Ts*&&... vs) const

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

Note: This function is part of the invocation policy implemented by this class

template<typename **Action**, typename **Continuation**, typename ...**Ts**>
inline bool **apply**(*Continuation* &&c, *launch* policy, *Ts*&&... vs) const

Note: This function is part of the invocation policy implemented by this class

template<typename **Action**, typename ...**Ts**>
inline bool **apply**(*launch* policy, *Ts*&&... vs) const

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*Continuation* &&c, *launch* policy, *Callback* &&cb, *Ts*&&... vs) const

Note: This function is part of the invocation policy implemented by this class

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

inline *hpx*::id_type **get_next_target**() const

Returns the locality which is anticipated to be used for the next async operation

Public Static Functions

static inline *std*::size_t **get_num_localities**()

Returns the number of associated localities for this distribution policy

Note: This function is part of the creation policy implemented by this class

template<typename **Action**>

struct **async_result**

#include <colocating_distribution_policy.hpp>

Note: This function is part of the invocation policy implemented by this class

Public Types

using **type** = *hpx*::*future*<typename *traits*::promise_local_result<typename *hpx*::*traits*::extract_action<*Action*>::remote_result_type>::type>

hpx/distribution_policies/default_distribution_policy.hpp

Defined in header hpx/distribution_policies/default_distribution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

Variables

static const *default_distribution_policy* **default_layout** = {}

A predefined instance of the default *distribution_policy*. It will represent the local locality and will place all items to create here.

struct **default_distribution_policy**

#include <default_distribution_policy.hpp> This class specifies the parameters for a simple distribution policy to use for creating (and evenly distributing) a given number of items on a given set of localities.

Public Functions

constexpr **default_distribution_policy**() = default

Default-construct a new instance of a default_distribution_policy. This policy will represent one locality (the local locality).

inline *default_distribution_policy* **operator()**(*std*::vector<id_type> const &locs) const

Create a new *default_distribution* policy representing the given set of localities.

Parameters **locs** – [in] The list of localities the new instance should represent

inline *default_distribution_policy* **operator()**(*std*::vector<id_type> &&locs) const

Create a new *default_distribution* policy representing the given set of localities.

Parameters **locs** – [in] The list of localities the new instance should represent

inline *default_distribution_policy* **operator()**(id_type const &loc) const

Create a new *default_distribution* policy representing the given locality

Parameters **loc** – [in] The locality the new instance should represent

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*hpx*::id_type> **create**(*Ts*&&... vs) const

Create one object on one of the localities associated by this policy instance

Note: This function is part of the placement policy implemented by this class

Parameters **vs** – [in] The arguments which will be forwarded to the constructor of the new object.

Returns A future holding the global address which represents the newly created object

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*std*::vector<bulk_locality_result>> **bulk_create**(*std*::size_t count, *Ts*&&... vs) const

Create multiple objects on the localities associated by this policy instance

Note: This function is part of the placement policy implemented by this class

Parameters

- **count** – [in] The number of objects to create
- **vs** – [in] The arguments which will be forwarded to the constructors of the new objects.

Returns A future holding the list of global addresses that represent the newly created objects

template<typename **Action**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async**(*launch* policy, *Ts*&&... vs) const

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

Note: This function is part of the invocation policy implemented by this class

template<typename **Action**, typename **Continuation**, typename ...**Ts**>

inline bool **apply**(*Continuation* &&c, *launch* policy, *Ts*&&... vs) const

> **Note:** This function is part of the invocation policy implemented by this class

template<typename **Action**, typename ...**Ts**>
inline bool **apply**(*threads*::*thread_priority* priority, *Ts*&&... vs) const

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*Continuation* &&c, *launch* policy, *Callback* &&cb, *Ts*&&... vs) const

> **Note:** This function is part of the invocation policy implemented by this class

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

inline *std*::size_t **get_num_localities**() const

Returns the number of associated localities for this distribution policy

> **Note:** This function is part of the creation policy implemented by this class

inline *hpx*::id_type **get_next_target**() const

Returns the locality which is anticipated to be used for the next async operation

template<typename **Action**>

struct **async_result**

#include <default_distribution_policy.hpp>

> **Note:** This function is part of the invocation policy implemented by this class

Public Types

using **type** = *hpx*::*future*<typename *traits*::promise_local_result<typename *hpx*::*traits*::extract_action<*Action*>::remote_result_type>::type>

hpx/distribution_policies/target_distribution_policy.hpp

Defined in header hpx/distribution_policies/target_distribution_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

Variables

static const *target_distribution_policy* **target** = { }

A predefined instance of the target_distribution_policy. It will represent the local locality and will place all items to create here.

struct **target_distribution_policy**

#include <target_distribution_policy.hpp> This class specifies the parameters for a simple distribution policy to use for creating (and evenly distributing) a given number of items on a given set of localities.

Public Functions

target_distribution_policy() = default

Default-construct a new instance of a target_distribution_policy. This policy will represent one locality (the local locality).

inline *target_distribution_policy* **operator()**(id_type const &id) const

Create a new target_distribution_policy representing the given locality

Parameters **loc** – [in] The locality the new instance should represent

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*hpx*::id_type> **create**(*Ts*&&... vs) const

Create one object on one of the localities associated by this policy instance

Note: This function is part of the placement policy implemented by this class

Parameters **vs** – [in] The arguments which will be forwarded to the constructor of the new object.

Returns A future holding the global address which represents the newly created object

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*std*::vector<bulk_locality_result>> **bulk_create**(*std*::size_t count, *Ts*&&... vs) const

Create multiple objects on the localities associated by this policy instance

Note: This function is part of the placement policy implemented by this class

Parameters
- **count** – [in] The number of objects to create
- **vs** – [in] The arguments which will be forwarded to the constructors of the new objects.

Returns A future holding the list of global addresses which represent the newly created objects

template<typename **Action**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async**(*launch* policy, *Ts*&&... vs) const

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

Note: This function is part of the invocation policy implemented by this class

template<typename **Action**, typename **Continuation**, typename ...**Ts**>
inline bool **apply**(*Continuation* &&c, *launch* policy, *Ts*&&... vs) const

Note: This function is part of the invocation policy implemented by this class

template<typename **Action**, typename ...**Ts**>
inline bool **apply**(*launch* policy, *Ts*&&... vs) const

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*Continuation* &&c, *launch* policy, *Callback* &&cb, *Ts*&&... vs) const

Note: This function is part of the invocation policy implemented by this class

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

inline *std*::size_t **get_num_localities**() const

Returns the number of associated localities for this distribution policy

Note: This function is part of the creation policy implemented by this class

inline *hpx*::id_type **get_next_target**() const

Returns the locality which is anticipated to be used for the next async operation

template<typename **Action**>

struct **async_result**

#include <target_distribution_policy.hpp>

Note: This function is part of the invocation policy implemented by this class

Public Types

using **type** = *hpx*::*future*<typename *traits*::promise_local_result<typename *hpx*::*traits*::extract_action<*Action*>::remote_result_type>::type>

hpx/distribution_policies/unwrapping_result_policy.hpp

Defined in header hpx/distribution_policies/unwrapping_result_policy.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

struct **unwrapping_result_policy**

#include <unwrapping_result_policy.hpp> This class is a distribution policy that can be using with actions that return futures. For those actions it is possible to apply certain optimizations if the action is invoked synchronously.

Public Functions

inline explicit **unwrapping_result_policy**(id_type const &id)

template<typename **Client**, typename **Stub**, typename **Data**>
inline explicit **unwrapping_result_policy**(client_base<*Client*, *Stub*, *Data*> const &client)

template<typename **Action**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async**(*launch* policy, *Ts*&&... vs) const

template<typename **Action**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async**(*launch*::sync_policy, *Ts*&&... vs) const

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline *async_result*<*Action*>::type **async_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

template<typename **Action**, typename **Continuation**, typename ...**Ts**>
inline bool **apply**(*Continuation* &&c, *launch* policy, *Ts*&&... vs) const

> **Note:** This function is part of the invocation policy implemented by this class

template<typename **Action**, typename ...**Ts**>
inline bool **apply**(*launch* policy, *Ts*&&... vs) const

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*Continuation* &&c, *launch* policy, *Callback* &&cb, *Ts*&&... vs) const

> **Note:** This function is part of the invocation policy implemented by this class

template<typename **Action**, typename **Callback**, typename ...**Ts**>
inline bool **apply_cb**(*launch* policy, *Callback* &&cb, *Ts*&&... vs) const

inline *hpx*::id_type const &**get_next_target**() const

template<typename **Action**>

struct **async_result**

Public Types

using **type** = typename *traits*::promise_local_result<typename *hpx*::*traits*::extract_action<*Action*>::remote_result_type>::type

executors_distributed

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/executors_distributed/distribution_policy_executor.hpp

Defined in header hpx/executors_distributed/distribution_policy_executor.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **execution**

Functions

template<typename **DistPolicy**>
distribution_policy_executor(*DistPolicy*&&) -> *distribution_policy_executor*<*std*::decay_t<*DistPolicy*>>

template<typename DistPolicy> HPX_DEPRECATED_V (1, 9, "hpx::parallel::execution::make_distribution_policy_executor is " "deprecated, use " "hpx::parallel::execution::distribution_policy_executor instead") distribution_policy

Create a new *distribution_policy_executor* from the given distribution_policy.

Parameters policy – The distribution_policy to create an executor from

template<typename **DistPolicy**>

class **distribution_policy_executor**

#include <distribution_policy_executor.hpp> A distribution_policy_executor creates groups of parallel execution agents that execute in threads implicitly created by the executor and placed on any of the associated localities.

Template Parameters DistPolicy – The distribution policy type for which an executor should be created. The expression *hpx::traits::is_distribution_policy_v<DistPolicy>* must evaluate to true.

init_runtime

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx::finalize, hpx::disconnect

Defined in header hpx/init.hpp[781].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

int **finalize**(double shutdown_timeout, double localwait = -1.0, *error_code* &ec = *throws*)

Main function to gracefully terminate the HPX runtime system.

The function *hpx::finalize* is the main way to (gracefully) exit any HPX application. It should be called from one locality only (usually the console) and it will notify all connected localities to finish execution. Only after all other localities have exited this function will return, allowing to exit the console locality as well.

During the execution of this function the runtime system will invoke all registered shutdown functions (see *hpx::init*) on all localities.

The default value (`-1.0`) will try to find a globally set timeout value (can be set as the configuration parameter `hpx.shutdown_timeout`), and if that is not set or `-1.0` as well, it will disable any timeout, each connected locality will wait for all existing HPX-threads to terminate.

The default value (`-1.0`) will try to find a globally set wait time value (can be set as the configuration parameter "hpx.finalize_wait_time"), and if this is not set or `-1.0` as well, it will disable any addition local wait time before proceeding.

This function will block and wait for all connected localities to exit before returning to the caller. It should be the last HPX-function called by any application.

Using this function is an alternative to *hpx::disconnect*, these functions do not need to be called both.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **`shutdown_timeout`** – This parameter allows to specify a timeout (in microseconds), specifying how long any of the connected localities should wait for pending tasks to be executed. After this timeout, all suspended HPX-threads will be aborted. Note, that this function will not abort any running HPX-threads. In any case the shutdown will not proceed as long as there is at least one pending/running HPX-thread.

[781] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/init_runtime/include/hpx/init.hpp

- **localwait** – This parameter allows to specify a local wait time (in microseconds) before the connected localities will be notified and the overall shutdown process starts.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will always return zero.

inline int **finalize**(*error_code* &ec = *throws*)

Main function to gracefully terminate the HPX runtime system.

The function *hpx::finalize* is the main way to (gracefully) exit any HPX application. It should be called from one locality only (usually the console) and it will notify all connected localities to finish execution. Only after all other localities have exited this function will return, allowing to exit the console locality as well.

During the execution of this function the runtime system will invoke all registered shutdown functions (see *hpx::init*) on all localities.

This function will block and wait for all connected localities to exit before returning to the caller. It should be the last HPX-function called by any application.

Using this function is an alternative to *hpx::disconnect*, these functions do not need to be called both.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will always return zero.

void **terminate**()

Terminate any application non-gracefully.

The function *hpx::terminate* is the non-graceful way to exit any application immediately. It can be called from any locality and will terminate all localities currently used by the application.

Note: This function will cause HPX to call `std::terminate()` on all localities associated with this application. If the function is called not from an HPX thread it will fail and return an error using the argument *ec*.

---

int **disconnect**(double shutdown_timeout, double localwait = -1.0, *error_code* &ec = *throws*)

Disconnect this locality from the application.

The function *hpx::disconnect* can be used to disconnect a locality from a running HPX application.

During the execution of this function the runtime system will invoke all registered shutdown functions (see *hpx::init*) on this locality.

The default value (`-1.0`) will try to find a globally set timeout value (can be set as the configuration parameter "hpx.shutdown_timeout"), and if that is not set or `-1.0` as well, it will disable any timeout, each connected locality will wait for all existing HPX-threads to terminate.

The default value (`-1.0`) will try to find a globally set wait time value (can be set as the configuration parameter `hpx.finalize_wait_time`), and if this is not set or `-1.0` as well, it will disable any addition local wait time before proceeding.

This function will block and wait for this locality to finish executing before returning to the caller. It should be the last HPX-function called by any locality being disconnected.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters

- **`shutdown_timeout`** – This parameter allows to specify a timeout (in microseconds), specifying how long this locality should wait for pending tasks to be executed. After this timeout, all suspended HPX-threads will be aborted. Note, that this function will not abort any running HPX-threads. In any case the shutdown will not proceed as long as there is at least one pending/running HPX-thread.
- **`localwait`** – This parameter allows to specify a local wait time (in microseconds) before the connected localities will be notified and the overall shutdown process starts.
- **`ec`** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will always return zero.

inline int **`disconnect`**(*error_code* &ec = *throws*)

Disconnect this locality from the application.

The function *hpx::disconnect* can be used to disconnect a locality from a running HPX application.

During the execution of this function the runtime system will invoke all registered shutdown functions (see *hpx::init*) on this locality.

This function will block and wait for this locality to finish executing before returning to the caller. It should be the last HPX-function called by any locality being disconnected.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters **`ec`** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will always return zero.

int **`stop`**(*error_code* &ec = *throws*)

Stop the runtime system.

This function will block and wait for this locality to finish executing before returning to the caller. It should be the last HPX-function called on every locality. This function should be used only if the runtime system was started using `hpx::start`.

Returns The function returns the value, which has been returned from the user supplied main HPX function (usually `hpx_main`).

hpx/hpx_init.hpp

Defined in header hpx/hpx_init.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

hpx::init

Defined in header hpx/init.hpp[782].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx_startup**

Variables

std::function<int(*hpx*::program_options::variables_map&)> const &**get_main_func**()

namespace **hpx**

Functions

inline int **init**(*std*::function<int(*hpx*::program_options::variables_map&)> f, int argc, char **argv, *init_params* const &params = *init_params*())

Main entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

[782] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/init_runtime/include/hpx/init.hpp

- **f** – [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- **argc** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **argv** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **params** – [in] The parameters to the *hpx::init* function (See documentation of hpx::init_params)

Returns The function returns the value, which has been returned from the user supplied `f`.

inline int **init**(*std*::function<int(int, char**)> f, int argc, char **argv, *init_params* const &params = *init_params*())

Main entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- **f** – [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- **argc** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **argv** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **params** – [in] The parameters to the *hpx::init* function (See documentation of hpx::init_params)

Returns The function returns the value, which has been returned from the user supplied `f`.

inline int **init**(int argc, char **argv, *init_params* const &params = *init_params*())

Main entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- **`argc`** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **`argv`** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **`params`** – [in] The parameters to the *hpx::init* function (See documentation of hpx::init_params)

Returns The function returns the value, which has been returned from the user supplied `f`.

inline int **`init`**(*std*::nullptr_t f, int argc, char **argv, *init_params* const &params = *init_params*())

Main entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- **`f`** – [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- **`argc`** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **`argv`** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **`params`** – [in] The parameters to the *hpx::init* function (See documentation of hpx::init_params)

Returns The function returns the value, which has been returned from the user supplied `f`.

inline int **init**(*init_params* const &params = *init_params*())

Main entry point for launching the HPX runtime system.

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

Note: The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc/argv`. If not command line arguments are passed, console mode is assumed.

Note: If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section 'HPX Command Line Options'.

Parameters **params** – [in] The parameters to the *hpx::init* function (See documentation of hpx::init_params)

Returns The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

hpx::init_params

Defined in header hpx/init.hpp[783].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

struct **init_params**

#include <hpx_init_params.hpp> Parameters used to initialize the HPX runtime through *hpx::init* and *hpx::start*.

Public Functions

inline **init_params**()

[783] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/init_runtime/include/hpx/init.hpp

Public Members

std::reference_wrapper<*hpx*::program_options::options_description const> **desc_cmdline** = *hpx*::local::detail::default_desc(HPX_APPLICATION_STRING)

std::vector<*std*::string> **cfg**

mutable *startup_function_type* **startup**

mutable *shutdown_function_type* **shutdown**

hpx::*runtime_mode* **mode** = ::*hpx*::*runtime_mode*::*default_*

hpx::*resource*::partitioner_mode **rp_mode** = ::*hpx*::*resource*::partitioner_mode::default_

hpx::*resource*::rp_callback_type **rp_callback**

hpx/hpx_start.hpp

Defined in header hpx/hpx_start.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

hpx::start

Defined in header hpx/init.hpp[784].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx_startup**

namespace **hpx**

Functions

inline bool **start**(*std*::function<int(*hpx*::program_options::variables_map&)> f, int argc, char **argv, *init_params* const &params = *init_params*())

Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

[784] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/init_runtime/include/hpx/init.hpp

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- **`f`** – [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- **`argc`** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **`argv`** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **`params`** – [in] The parameters to the *hpx::start* function (See documentation of hpx::init_params)

Returns The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

inline bool **`start`**(*std*::function<int(int, char**)> f, int argc, char **argv, *init_params* const &params = *init_params*())

Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- **`f`** – [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.

- **argc** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **argv** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **params** – [in] The parameters to the *hpx::start* function (See documentation of hpx::init_params)

Returns The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

inline bool **start**(int argc, char **argv, *init_params* const &params = *init_params*())

Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- **argc** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **argv** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **params** – [in] The parameters to the *hpx::start* function (See documentation of hpx::init_params)

Returns The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

inline bool **start**(*std*::nullptr_t f, int argc, char **argv, *init_params* const &params = *init_params*())

Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Note: If the parameter `mode` is not given (defaulted), the created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. Otherwise it will be executed as specified by the parameter`mode`.

Parameters

- **`f`** – [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- **`argc`** – [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- **`argv`** – [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- **`params`** – [in] The parameters to the *hpx::start* function (See documentation of hpx::init_params)

Returns The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

inline bool **`start`**(*init_params* const &params = *init_params*())

Main non-blocking entry point for launching the HPX runtime system.

This is a simplified main, non-blocking entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings). It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and `hpx::stop` to synchronize with the runtime system's execution.

Note: The created runtime system instance will be executed in console or worker mode depending on the command line arguments passed in `argc`/`argv`. If not command line arguments are passed, console mode is assumed.

Note: If no command line arguments are passed the HPX runtime system will not support any of the default command line options as described in the section 'HPX Command Line Options'.

Parameters **`params`** – [in] The parameters to the *hpx::start* function (See documentation of hpx::init_params)

Returns The function returns *true* if command line processing succeeded and the runtime system was started successfully. It will return *false* otherwise.

hpx::suspend, hpx::resume

Defined in header hpx/init.hpp[785].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

int **suspend**(*error_code* &ec = *throws*)

Suspend the runtime system.

The function *hpx::suspend* is used to suspend the HPX runtime system. It can only be used when running HPX on a single locality. It will block waiting for all thread pools to be empty. This function only be called when the runtime is running, or already suspended in which case this function will do nothing.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will always return zero.

int **resume**(*error_code* &ec = *throws*)

Resume the HPX runtime system.

The function *hpx::resume* is used to resume the HPX runtime system. It can only be used when running HPX on a single locality. It will block waiting for all thread pools to be resumed. This function only be called when the runtime suspended, or already running in which case this function will do nothing.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns This function will always return zero.

[785] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/init_runtime/include/hpx/init.hpp

naming_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/naming_base/unmanaged.hpp

Defined in header hpx/naming_base/unmanaged.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

hpx::id_type **unmanaged**(*hpx*::id_type const &id)

The helper function *hpx::unmanaged* can be used to generate a global identifier which does not participate in the automatic garbage collection.

Note: This function allows to apply certain optimizations to the process of memory management in HPX. It however requires the user to take full responsibility for keeping the referenced objects alive long enough.

Parameters **id** – [in] The id to generated the unmanaged global id from This parameter can be itself a managed or a unmanaged global id.

Returns This function returns a new global id referencing the same object as the parameter *id*. The only difference is that the returned global identifier does not participate in the automatic garbage collection.

namespace **naming**

parcelset

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parcelset/connection_cache.hpp

Defined in header hpx/parcelset/connection_cache.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parcelset/message_handler_fwd.hpp

Defined in header hpx/parcelset/message_handler_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parcelset/parcelhandler.hpp

Defined in header hpx/parcelset/parcelhandler.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parcelset/parcelset_fwd.hpp

Defined in header hpx/parcelset/parcelset_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

parcelset_base

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parcelset_base/parcelport.hpp

Defined in header hpx/parcelset_base/parcelport.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parcelset_base/parcelset_base_fwd.hpp

Defined in header hpx/parcelset_base/parcelset_base_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_PARCELPORT_BACKGROUND_MODE_ENUM_DEPRECATION_MSG

namespace **hpx**

namespace **parcelset**

Typedefs

using **parcel_write_handler_type** = *hpx*::*function*<void(*std*::error_code const&, *parcelset*::parcel const&)>

The type of the function that can be registered as a parcel write handler using the function *hpx::set_parcel_write_handler*.

Note: A parcel write handler is a function which is called by the parcel layer whenever a parcel has been sent by the underlying networking library and if no explicit parcel handler function was specified for the parcel.

Enums

enum class **parcelport_background_mode** : *std*::uint8_t

Type of background work to perform.

Values:

enumerator **flush_buffers**

perform buffer flush operations

enumerator **send**

perform send operations (includes buffer flush)

enumerator **receive**

perform receive operations

enumerator **all**

perform all operations

Functions

inline bool **operator&**(*parcelport_background_mode* lhs, *parcelport_background_mode* rhs)

char const ***get_parcelport_background_mode_name**(*parcelport_background_mode* mode)

Variables

parcel **empty_parcel**

constexpr *parcelport_background_mode* **parcelport_background_mode_flush_buffers** = *parcelport_background_mode*::*flush_buffers*

constexpr *parcelport_background_mode* **parcelport_background_mode_send** = *parcelport_background_mode*::*send* constexpr *parcelport_background_mode* **parcelport_background_mode_receive** = *parcelport_background_mode*::*receive*

constexpr *parcelport_background_mode* **parcelport_background_mode_all** = *parcelport_background_mode*::*all*

hpx/parcelset_base/set_parcel_write_handler.hpp

Defined in header hpx/parcelset_base/set_parcel_write_handler.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

performance_counters

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/performance_counters/counter_creators.hpp

Defined in header hpx/performance_counters/counter_creators.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **performance_counters**

Functions

bool **default_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

Default discovery function for performance counters; to be registered with the counter types. It will pass the counter_info and the *error_code* to the supplied function.

bool **locality_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/total)/<instancename>

bool **locality_pool_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/pool#<pool_name>/total)/<instancename>

bool **locality0_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

Default discoverer function for AGAS performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#0/total}/<instancename>

bool **locality_thread_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/worker-thread#<threadnum>)/<instancename>

bool **locality_pool_thread_counter_discoverer**(*counter_info* const &info, *discover_counter_func* const &f, *discover_counters_mode* mode, *error_code* &ec)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#<locality_id>/pool#<poolname>/thread#<threadnum>}/<instancename>

bool **locality_pool_thread_no_total_counter_discoverer**(*counter_info* const &info, *discover_counter_func* const &f, *discover_counters_mode* mode, *error_code* &ec)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#<locality_id>/pool#<poolname>/thread#<threadnum>}/<instancename>

This is essentially the same as above just that locality#*/total is not supported.

bool **locality_numa_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/numa-node#<threadnum>)/<instancename>

naming::gid_type **locality_raw_counter_creator**(*counter_info* const&, *hpx*::*function*<*std*::int64_t(bool)> const&, *error_code*&)

Creation function for raw counters. The passed function is encapsulating the actual value to monitor. This function checks the validity of the supplied counter name, it has to follow the scheme:

/<objectname>(locality#<locality_id>/total)/<instancename>

naming::gid_type **locality_raw_values_counter_creator**(*counter_info* const&, *hpx*::*function*<*std*::vector<*std*::int64_t>(bool)> const&, *error_code*&)

naming::gid_type **agas_raw_counter_creator**(*counter_info* const&, *error_code*&, char const*const)

Creation function for raw counters. The passed function is encapsulating the actual value to monitor. This function checks the validity of the supplied counter name, it has to follow the scheme:

/agas(<objectinstance>/total)/<instancename>

bool **agas_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/agas(<objectinstance>/total)/<instancename>

naming::gid_type **local_action_invocation_counter_creator**(*counter_info* const&, *error_code*&)

bool **local_action_invocation_counter_discoverer**(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)

hpx/performance_counters/counters.hpp

Defined in header hpx/performance_counters/counters.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **performance_counters**

Typedefs

typedef *hpx*::*function*<*naming*::gid_type(*counter_info* const&, *error_code*&)> **create_counter_func**

This declares the type of a function, which will be called by HPX whenever a new performance counter instance of a particular type needs to be created.

typedef *hpx*::*function*<bool(*counter_info* const&, *error_code*&)> **discover_counter_func**

This declares a type of a function, which will be passed to a *discover_counters_func* in order to be called for each discovered performance counter instance.

typedef *hpx*::*function*<bool(*counter_info* const&, *discover_counter_func* const&, *discover_counters_mode*, *error_code*&)> **discover_counters_func**

This declares the type of a function, which will be called by HPX whenever it needs to discover all performance counter instances of a particular type.

Enums

enum class **counter_type**

Values:

enumerator **text**

text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

enumerator **raw**

raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

enumerator **monotonically_increasing**

monotonically_increasing shows the cumulatively accumulated observed value. It does not deliver an average.

Formula: None. Shows cumulatively accumulated data as collected. Average: None Type: Instantaneous

enumerator **average_base**

average_base is used as the base data (denominator) in the computation of time or count averages for the *counter_type::average_count* and *counter_type::average_timer* counter types. This counter type collects the last observed value only.

Formula: None. This counter uses raw data in factional calculations without delivering an output. Average: SUM (N) / x Type: Instantaneous

enumerator **average_count**

average_count shows how many items are processed, on average, during an operation. Counters of this type display a ratio of the items processed (such as bytes sent) to the number of operations completed. The ratio is calculated by comparing the number of items processed during the last interval to the number of operations completed during the last interval.

Formula: (N1 - N0) / (D1 - D0), where the numerator (N) represents the number of items processed during the last sample interval, and the denominator (D) represents the number of operations completed during the last two sample intervals. Average: (Nx - N0) / (Dx - D0) Type: Average

enumerator **aggregating**

aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: F(Nx)

enumerator **average_timer**

average_timer measures the average time it takes to complete a process or operation. Counters of this type display a ratio of the total elapsed time of the sample interval to the number of processes or operations completed during that time. This counter type measures time in ticks of the system clock. The variable F represents the number of ticks per second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: ((N1 - N0) / F) / (D1 - D0), where the numerator (N) represents the number of ticks counted during the last sample interval, the variable F represents the frequency of the ticks, and the denominator (D) represents the number of operations completed during the last sample interval. Average: ((Nx - N0) / F) / (Dx - D0) Type: Average

enumerator **elapsed_time**

elapsed_time shows the total time between when the component or process started and the time when this value is calculated. The variable F represents the number of time units that elapse in one second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: (D0 - N0) / F, where the nominator (D) represents the current time, the numerator (N) represents the time the object was started, and the variable F represents the number of time units that elapse in one second. Average: (Dx - N0) / F Type: Difference

enumerator **histogram**

histogram exposes a histogram of the measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a counter_value. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

The first three values in the returned array represent the lower and upper boundaries, and the size of the histogram buckets. All remaining values in the returned array represent the number of measurements for each of the buckets in the histogram.

enumerator **raw_values**

raw_values exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a counter_value. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

enumerator **text**

enumerator **raw**

enumerator **monotonically_increasing**

enumerator **average_base**

enumerator **average_count**

enumerator **aggregating**

enumerator **average_timer**

enumerator **elapsed_time**

enumerator **histogram**

enumerator **raw_values**

raw_values counter exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a counter_value. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

enum class **counter_status**

Status and error codes used by the functions related to performance counters.

Values:

enumerator **valid_data**

No error occurred, data is valid.

enumerator **new_data**

Data is valid and different from last call.

enumerator **invalid_data**

Some error occurred, data is not value.

enumerator **already_defined**

The type or instance already has been defined.

enumerator **counter_unknown**

The counter instance is unknown.

enumerator **counter_type_unknown**

The counter type is unknown.

enumerator **generic_error**

A unknown error occurred.

enumerator **valid_data**

enumerator **new_data**

enumerator **invalid_data**

enumerator **already_defined**

enumerator **counter_unknown**

enumerator **counter_type_unknown**

enumerator **generic_error**

Functions

inline *std*::string &**ensure_counter_prefix**(*std*::string &name)

inline *std*::string **ensure_counter_prefix**(*std*::string const &counter)

inline *std*::string &**remove_counter_prefix**(*std*::string &name)

inline *std*::string **remove_counter_prefix**(*std*::string const &counter)

char const ***get_counter_type_name**(*counter_type* state)

Return the readable name of a given counter type.

inline bool **status_is_valid**(*counter_status* s)

inline *counter_status* **add_counter_type**(*counter_info* const &info, *error_code* &ec)

inline *hpx*::id_type **get_counter**(*std*::string const &name, *error_code* &ec)

inline *hpx*::id_type **get_counter**(*counter_info* const &info, *error_code* &ec)

Variables

constexpr const char **counter_prefix**[] = "/counters"

constexpr *std*::size_t **counter_prefix_len** = *std*::size(*counter_prefix*) - 1

struct **counter_info**

Public Functions

inline explicit **counter_info**(*counter_type* type = *counter_type*::*raw*)

inline explicit **counter_info**(*std*::string const &name)

inline **counter_info**(*counter_type* type, *std*::string const &name, *std*::string const &helptext = "", *std*::uint32_t version = HPX_PERFORMANCE_COUNTER_V1, *std*::string const &uom = "")

Public Members

counter_type **type_**

The type of the described counter.

std::uint32_t **version_**

The version of the described counter using the 0xMMmmSSSS scheme

counter_status **status_**

The status of the counter object.

std::string **fullname_**

The full name of this counter.

std::string **helptext_**

The full descriptive text for this counter.

std::string **unit_of_measure_**

The unit of measure for this counter.

Private Functions

void **serialize**(*serialization*::output_archive &ar, unsigned int) const

void **serialize**(*serialization*::input_archive &ar, unsigned int)

Friends

friend class hpx::serialization::access

struct **counter_path_elements** : public *hpx*::*performance_counters*::*counter_type_path_elements*

#include <counters.hpp> A *counter_path_elements* holds the elements of a full name for a counter instance. Generally, a full name of a counter instance has the structure:

/objectname{parentinstancename::parentindex/instancename#instanceindex} /countername#parameters

i.e. /queue{localityprefix/thread#2}/length

Public Types

using **base_type** = *counter_type_path_elements*

Public Functions

inline **counter_path_elements**()

inline **counter_path_elements**(*std*::string const &objectname, *std*::string const &countername, *std*::string const &parameters, *std*::string const &parentname, *std*::string const &instancename, *std*::int64_t parentindex = -1, *std*::int64_t instanceindex = -1, bool parentinstance_is_basename = false)

inline **counter_path_elements**(*std*::string const &objectname, *std*::string const &countername, *std*::string const &parameters, *std*::string const &parentname, *std*::string const &instancename, *std*::string const &subinstancename, *std*::int64_t parentindex = -1, *std*::int64_t instanceindex = -1, *std*::int64_t subinstanceindex = -1, bool parentinstance_is_basename = false)

Public Members

std::string **parentinstancename_**

the name of the parent instance

std::string **instancename_**

the name of the object instance

std::string **subinstancename_**

the name of the object sub-instance

std::int64_t **parentinstanceindex_**

the parent instance index

std::int64_t **instanceindex_**

the instance index

std::int64_t **subinstanceindex_**

the sub-instance index

bool **parentinstance_is_basename_**

the parentinstancename_

Private Functions

void **serialize**(*serialization*::output_archive &ar, unsigned int)

void **serialize**(*serialization*::input_archive &ar, unsigned int)

Friends

friend class hpx::serialization::access

struct **counter_type_path_elements**

#include <counters.hpp> A *counter_type_path_elements* holds the elements of a full name for a counter type. Generally, a full name of a counter type has the structure:

/objectname/countername

i.e. /queue/length

Subclassed by *hpx::performance_counters::counter_path_elements*

Public Functions

counter_type_path_elements() = default

inline **counter_type_path_elements**(*std*::string const &objectname, *std*::string const &countername, *std*::string const &parameters)

Public Members

std::string **objectname_**

the name of the performance object

std::string **countername_**

contains the counter name

std::string **parameters_**

optional parameters for the counter instance

Protected Functions

void **serialize**(*serialization*::output_archive &ar, unsigned int) const

void **serialize**(*serialization*::input_archive &ar, unsigned int)

Friends

friend class hpx::serialization::access

hpx/performance_counters/counters_fwd.hpp

Defined in header hpx/performance_counters/counters_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_COUNTER_TYPE_UNSCOPED_ENUM_DEPRECATION_MSG

HPX_COUNTER_STATUS_UNSCOPED_ENUM_DEPRECATION_MSG

HPX_PERFORMANCE_COUNTER_V1

HPX_DISCOVER_COUNTERS_MODE_UNSCOPED_ENUM_DEPRECATION_MSG

namespace **hpx**

namespace **performance_counters**

Enums

enum class **`counter_type`**

Values:

enumerator **`text`**

text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

enumerator **`raw`**

raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

enumerator **`monotonically_increasing`**

monotonically_increasing shows the cumulatively accumulated observed value. It does not deliver an average.

Formula: None. Shows cumulatively accumulated data as collected. Average: None Type: Instantaneous

enumerator **`average_base`**

average_base is used as the base data (denominator) in the computation of time or count averages for the *counter_type::average_count* and *counter_type::average_timer* counter types. This counter type collects the last observed value only.

Formula: None. This counter uses raw data in factional calculations without delivering an output. Average: SUM (N) / x Type: Instantaneous

enumerator **`average_count`**

average_count shows how many items are processed, on average, during an operation. Counters of this type display a ratio of the items processed (such as bytes sent) to the number of operations completed. The ratio is calculated by comparing the number of items processed during the last interval to the number of operations completed during the last interval.

Formula: (N1 - N0) / (D1 - D0), where the numerator (N) represents the number of items processed during the last sample interval, and the denominator (D) represents the number of operations completed during the last two sample intervals. Average: (Nx - N0) / (Dx - D0) Type: Average

enumerator **`aggregating`**

aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: F(Nx)

enumerator **`average_timer`**

average_timer measures the average time it takes to complete a process or operation. Counters of this type display a ratio of the total elapsed time of the sample interval to the number of processes or operations completed during that time. This counter type measures time in ticks of the system clock. The variable F represents the number of ticks per second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: ((N1 - N0) / F) / (D1 - D0), where the numerator (N) represents the number of ticks counted during the last sample interval, the variable F represents the frequency of the ticks, and the denominator (D) represents the number of operations completed during the last sample interval. Average: ((Nx - N0) / F) / (Dx - D0) Type: Average

enumerator **elapsed_time**

elapsed_time shows the total time between when the component or process started and the time when this value is calculated. The variable F represents the number of time units that elapse in one second. The value of F is factored into the equation so that the result is displayed in seconds.

Formula: (D0 - N0) / F, where the nominator (D) represents the current time, the numerator (N) represents the time the object was started, and the variable F represents the number of time units that elapse in one second. Average: (Dx - N0) / F Type: Difference

enumerator **histogram**

histogram exposes a histogram of the measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a counter_value. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

The first three values in the returned array represent the lower and upper boundaries, and the size of the histogram buckets. All remaining values in the returned array represent the number of measurements for each of the buckets in the histogram.

enumerator **raw_values**

raw_values exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a counter_value. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

enumerator **text**

enumerator **raw**

enumerator **monotonically_increasing**

enumerator **average_base**

enumerator **average_count**

enumerator **aggregating**

enumerator **average_timer**

enumerator **elapsed_time**

enumerator **histogram**

enumerator **raw_values**

raw_values counter exposes an array of measured values instead of a single value as many of the other counter types. Counters of this type expose a *counter_value_array* instead of a counter_value. Those will also not implement the *get_counter_value()* functionality. The results are exposed through a separate *get_counter_values_array()* function.

enum class **counter_status**

Values:

enumerator **valid_data**

No error occurred, data is valid.

enumerator **new_data**

Data is valid and different from last call.

enumerator **invalid_data**

Some error occurred, data is not value.

enumerator **already_defined**

The type or instance already has been defined.

enumerator **counter_unknown**

The counter instance is unknown.

enumerator **counter_type_unknown**

The counter type is unknown.

enumerator **generic_error**

A unknown error occurred.

enumerator **valid_data**

enumerator **new_data**

enumerator **invalid_data**

enumerator **already_defined**

enumerator **counter_unknown**

enumerator **counter_type_unknown**

enumerator **generic_error**

enum class **discover_counters_mode**

Values:

enumerator **minimal**

enumerator **full**

Functions

inline constexpr bool **operator<**(*counter_type* lhs, *counter_type* rhs) noexcept

inline constexpr bool **operator>**(*counter_type* lhs, *counter_type* rhs) noexcept

std::ostream &**operator<<**(*std*::ostream &os, *counter_status* rhs)

counter_status **get_counter_type_name**(*counter_type_path_elements* const &path, *std*::string &result, *error_code* &ec = *throws*)

Create a full name of a counter type from the contents of the given counter_type_path_elements instance.The generated counter type name will not contain any parameters.

counter_status **get_full_counter_type_name**(*counter_type_path_elements* const &path, *std*::string &result, *error_code* &ec = *throws*)

Create a full name of a counter type from the contents of the given counter_type_path_elements instance. The generated counter type name will contain all parameters.

counter_status **get_counter_name**(*counter_path_elements* const &path, *std*::string &result, *error_code* &ec = *throws*)

Create a full name of a counter from the contents of the given counter_path_elements instance.

counter_status **get_counter_instance_name**(*counter_path_elements* const &path, *std*::string &result, *error_code* &ec = *throws*)

Create a name of a counter instance from the contents of the given counter_path_elements instance.

counter_status **get_counter_type_path_elements**(*std*::string const &name, *counter_type_path_elements* &path, *error_code* &ec = *throws*)

Fill the given counter_type_path_elements instance from the given full name of a counter type.

counter_status **get_counter_path_elements**(*std*::string const &name, *counter_path_elements* &path, *error_code* &ec = *throws*)

Fill the given counter_path_elements instance from the given full name of a counter.

counter_status **get_counter_name**(*std*::string const &name, *std*::string &countername, *error_code* &ec = *throws*)

Return the canonical counter instance name from a given full instance name.

counter_status **get_counter_type_name**(*std*::string const &name, *std*::string &type_name, *error_code* &ec = *throws*)

Return the canonical counter type name from a given (full) instance name.

HPX_DEPRECATED_V (1, 9, HPX_DISCOVER_COUNTERS_MODE_UNSCOPED_ENUM_DEPRECATION_MSG) inline const expr discover_counters_

counter_status **complement_counter_info**(*counter_info* &info, *counter_info* const &type_info, *error_code* &ec = *throws*)

Complement the counter info if parent instance name is missing.

counter_status **complement_counter_info**(*counter_info* &info, *error_code* &ec = *throws*)

counter_status **add_counter_type**(*counter_info* const &info, *create_counter_func* const &create_counter, *discover_counters_func* const &discover_counters, *error_code* &ec = *throws*)

counter_status **discover_counter_types**(*discover_counter_func* const &discover_counter, *discover_counters_mode* mode = *discover_counters_mode*::*minimal*, *error_code* &ec = *throws*)

Call the supplied function for each registered counter type.

counter_status **discover_counter_types**(*std*::vector<*counter_info*> &counters, *discover_counters_mode* mode = *discover_counters_mode*::*minimal*, *error_code* &ec = *throws*)

Return a list of all available counter descriptions.

counter_status **discover_counter_type**(*std*::string const &name, *discover_counter_func* const &discover_counter, *discover_counters_mode* mode = *discover_counters_mode*::*minimal*, *error_code* &ec = *throws*)

Call the supplied function for the given registered counter type.

counter_status **discover_counter_type**(*counter_info* const &info, *discover_counter_func* const &discover_counter, *discover_counters_mode* mode = *discover_counters_mode*::*minimal*, *error_code* &ec = *throws*)

counter_status **discover_counter_type**(*std*::string const &name, *std*::vector<*counter_info*> &counters, *discover_counters_mode* mode = *discover_counters_mode*::*minimal*, *error_code* &ec = *throws*)

Return a list of matching counter descriptions for the given registered counter type.

counter_status **discover_counter_type**(*counter_info* const &info, *std*::vector<*counter_info*> &counters, *discover_counters_mode* mode = *discover_counters_mode*::*minimal*, *error_code* &ec = *throws*)

bool **expand_counter_info**(*counter_info* const&, *discover_counter_func* const&, *error_code*&)

call the supplied function will all expanded versions of the supplied counter info.

This function expands all locality#* and worker-thread#* wild cards only.

counter_status **remove_counter_type**(*counter_info* const &info, *error_code* &ec = *throws*)

Remove an existing counter type from the (local) registry.

Note: This doesn't remove existing counters of this type, it just inhibits defining new counters using this type.

counter_status **get_counter_type**(*std*::string const &name, *counter_info* &info, *error_code* &ec = *throws*)

Retrieve the counter type for the given counter name from the (local) registry.

hpx::*future*<*hpx*::id_type> **get_counter_async**(*std*::string name, *error_code* &ec = *throws*)

Get the global id of an existing performance counter, if the counter does not exist yet, the function attempts to create the counter based on the given counter name.

hpx::*future*<*hpx*::id_type> **get_counter_async**(*counter_info* const &info, *error_code* &ec = *throws*)

Get the global id of an existing performance counter, if the counter does not exist yet, the function attempts to create the counter based on the given counter info.

void **get_counter_infos**(*counter_info* const &info, *counter_type* &type, *std*::string &helptext, *std*::uint32_t &version, *error_code* &ec = *throws*)

Retrieve the meta data specific for the given counter instance.

void **get_counter_infos**(*std*::string name, *counter_type* &type, *std*::string &helptext, *std*::uint32_t &version, *error_code* &ec = *throws*)

Retrieve the meta data specific for the given counter instance.

Variables

constexpr *counter_type* **counter_text** = *counter_type*::*text*

constexpr *counter_type* **counter_raw** = *counter_type*::*raw*

constexpr *counter_type* **counter_monotonically_increasing** = *counter_type*::*monotonically_increasing*

constexpr *counter_type* **counter_average_base** = *counter_type*::*average_base*

constexpr *counter_type* **counter_average_count** = *counter_type*::*average_count*

constexpr *counter_type* **counter_aggregating** = *counter_type*::*aggregating*

constexpr *counter_type* **counter_average_timer** = *counter_type*::*average_timer*

constexpr *counter_type* **counter_elapsed_time** = *counter_type*::*elapsed_time*

constexpr *counter_type* **counter_raw_values** = *counter_type*::*raw_values*

constexpr *counter_type* **counter_histogram** = *counter_type*::*histogram*

constexpr *counter_status* **status_valid_data** = *counter_status*::*valid_data*

constexpr *counter_status* **status_new_data** = *counter_status*::*new_data* constexpr *counter_status* **status_invalid_data** = *counter_status*::*invalid_data*

constexpr *counter_status* **status_already_defined** = *counter_status*::*already_defined*

constexpr *counter_status* **status_counter_unknown** = *counter_status*::*counter_unknown*

constexpr *counter_status* **status_counter_type_unknown** = *counter_status*::*counter_type_unknown*

constexpr *counter_status* **status_generic_error** = *counter_status*::*generic_error*

struct **counter_value**

Public Functions

inline **counter_value**(*std*::int64_t value = 0, *std*::int64_t scaling = 1, bool scale_inverse = false)

template<typename **T**>
inline *T* **get_value**(*error_code* &ec = *throws*) const

Retrieve the 'real' value of the *counter_value*, converted to the requested type *T*.

Public Members

std::uint64_t **time_**

The local time when data was collected.

std::uint64_t **count_**

The invocation counter for the data.

std::int64_t **value_**

The current counter value.

std::int64_t **scaling_**

The scaling of the current counter value.

counter_status **status_**

The status of the counter value.

bool **scale_inverse_**

If true, value_ needs to be divided by scaling_, otherwise it has to be multiplied.

Private Functions

void **serialize**(*serialization*::output_archive &ar, const unsigned int) const

void **serialize**(*serialization*::input_archive &ar, const unsigned int)

Friends

friend class hpx::serialization::access

struct **counter_values_array**

Public Functions

inline **counter_values_array**(*std*::int64_t scaling = 1, bool scale_inverse = false)

inline **counter_values_array**(*std*::vector<*std*::int64_t> &&values, *std*::int64_t scaling = 1, bool scale_inverse = false)

inline **counter_values_array**(*std*::vector<*std*::int64_t> const &values, *std*::int64_t scaling = 1, bool scale_inverse = false)

template<typename **T**>
inline *T* **get_value**(*std*::size_t index, *error_code* &ec = *throws*) const

Retrieve the 'real' value of the *counter_value*, converted to the requested type *T*.

Public Members

std::uint64_t **time_**

The local time when data was collected.

std::uint64_t **count_**

The invocation counter for the data.

std::vector<*std*::int64_t> **values_**

The current counter values.

std::int64_t **scaling_**

The scaling of the current counter values.

counter_status **status_**

The status of the counter value.

bool **scale_inverse_**

If true, value_ needs to be divided by scaling_, otherwise it has to be multiplied.

Private Functions

void **serialize**(*serialization*::output_archive &ar, const unsigned int) const

void **serialize**(*serialization*::input_archive &ar, const unsigned int)

Friends

friend class hpx::serialization::access

hpx/performance_counters/manage_counter_type.hpp

Defined in header hpx/performance_counters/manage_counter_type.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **performance_counters**

Functions

counter_status **install_counter_type**(*std*::string const &name, *hpx*::*function*<*std*::int64_t(bool)> const &counter_value, *std*::string const &helptext = "", *std*::string const &uom = "", *counter_type* type = *counter_type*::*raw*, *error_code* &ec = *throws*)

Install a new generic performance counter type in a way, which will uninstall it automatically during shutdown.

The function *install_counter_type* will register a new generic counter type based on the provided function. The counter type will be automatically unregistered during system shutdown. Any consumer querying any instance of this this counter type will cause the provided function to be called and the returned value to be exposed as the counter value.

The counter type is registered such that there can be one counter instance per locality. The expected naming scheme for the counter instances is: `'/objectname{locality#<*>/total}/countername'` where '<*>' is a zero based integer identifying the locality the counter is created on.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: The counter type registry is a locality based service. You will have to register each counter type on every locality where a corresponding performance counter will be created.

Parameters

- **name** – [in] The global virtual name of the counter type. This name is expected to have the format /objectname/countername.

- **counter_value** – [in] The function to call whenever the counter value is requested by a consumer.
- **helptext** – [in, optional] A longer descriptive text shown to the user to explain the nature of the counters created from this type.
- **uom** – [in] The unit of measure for the new performance counter type.
- **type** – [in] Type for the new performance counter type.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns If successful, this function returns *valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*).

counter_status **install_counter_type**(*std*::string const &name, *hpx*::*function*<*std*::vector<*std*::int64_t>(bool)> const &counter_value, *std*::string const &helptext = "", *std*::string const &uom = "", *error_code* &ec = *throws*)

Install a new generic performance counter type returning an array of values in a way, that will uninstall it automatically during shutdown.

The function *install_counter_type* will register a new generic counter type that returns an array of values based on the provided function. The counter type will be automatically unregistered during system shutdown. Any consumer querying any instance of this this counter type will cause the provided function to be called and the returned array value to be exposed as the counter value.

The counter type is registered such that there can be one counter instance per locality. The expected naming scheme for the counter instances is: `'/objectname{locality#<*>/total}/countername'` where '<*>' is a zero based integer identifying the locality the counter is created on.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: The counter type registry is a locality based service. You will have to register each counter type on every locality where a corresponding performance counter will be created.

Parameters

- **name** – [in] The global virtual name of the counter type. This name is expected to have the format /objectname/countername.
- **counter_value** – [in] The function to call whenever the counter value (array of values) is requested by a consumer.
- **helptext** – [in, optional] A longer descriptive text shown to the user to explain the nature of the counters created from this type.
- **uom** – [in] The unit of measure for the new performance counter type.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns If successful, this function returns *valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*).

void **install_counter_type**(*std*::string const &name, *counter_type* type, *error_code* &ec = *throws*)

Install a new performance counter type in a way, which will uninstall it automatically during shutdown.

The function *install_counter_type* will register a new counter type based on the provided *counter_type_info*. The counter type will be automatically unregistered during system shutdown.

Note: The counter type registry is a locality based service. You will have to register each counter type on every locality where a corresponding performance counter will be created.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **name** – [in] The global virtual name of the counter type. This name is expected to have the format /objectname/countername.
- **type** – [in] The type of the counters of this counter_type.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns If successful, this function returns *valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*).

counter_status **install_counter_type**(*std*::string const &name, *counter_type* type, *std*::string const &helptext, *std*::string const &uom = "", *std*::uint32_t version = HPX_PERFORMANCE_COUNTER_V1, *error_code* &ec = *throws*)

Install a new performance counter type in a way, which will uninstall it automatically during shutdown.

The function *install_counter_type* will register a new counter type based on the provided *counter_type_info*. The counter type will be automatically unregistered during system shutdown.

Note: The counter type registry is a locality based service. You will have to register each counter type on every locality where a corresponding performance counter will be created.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- **name** – [in] The global virtual name of the counter type. This name is expected to have the format /objectname/countername.
- **type** – [in] The type of the counters of this counter_type.
- **helptext** – [in] A longer descriptive text shown to the user to explain the nature of the counters created from this type.
- **uom** – [in] The unit of measure for the new performance counter type.
- **version** – [in] The version of the counter type. This is currently expected to be set to HPX_PERFORMANCE_COUNTER_V1.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns If successful, this function returns *valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*).

counter_status **install_counter_type**(*std*::string const &name, *counter_type* type, *std*::string const &helptext, *create_counter_func* const &create_counter, *discover_counters_func* const &discover_counters, *std*::uint32_t version = HPX_PERFORMANCE_COUNTER_V1, *std*::string const &uom = "", *error_code* &ec = *throws*)

Install a new generic performance counter type in a way, which will uninstall it automatically during shutdown.

The function *install_counter_type* will register a new generic counter type based on the provided *counter_type_info*. The counter type will be automatically unregistered during system shutdown.

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: The counter type registry is a locality based service. You will have to register each counter type on every locality where a corresponding performance counter will be created.

Parameters

- **name** – [in] The global virtual name of the counter type. This name is expected to have the format /objectname/countername.
- **type** – [in] The type of the counters of this counter_type.
- **helptext** – [in] A longer descriptive text shown to the user to explain the nature of the counters created from this type.
- **version** – [in] The version of the counter type. This is currently expected to be set to HPX_PERFORMANCE_COUNTER_V1.
- **create_counter** – [in] The function which will be called to create a new instance of this counter type.
- **discover_counters** – [in] The function will be called to discover counter instances which can be created.
- **uom** – [in] The unit of measure of the counter type (default: “”)
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns If successful, this function returns *valid_data*, otherwise it will either throw an exception or return an error_code from the enum *counter_status* (also, see note related to parameter *ec*).

hpx/performance_counters/registry.hpp

Defined in header hpx/performance_counters/registry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **performance_counters**

class **registry**

Public Functions

registry() = default

void **clear**()

Reset registry by deleting all stored counter types.

counter_status **add_counter_type**(*counter_info* const &info, *create_counter_func* const &create_counter, *discover_counters_func* const &discover_counters, *error_code* &ec = *throws*)

Add a new performance counter type to the (local) registry.

counter_status **discover_counter_types**(*discover_counter_func* discover_counter, *discover_counters_mode* mode, *error_code* &ec = *throws*)

Call the supplied function for all registered counter types.

counter_status **discover_counter_type**(*std*::string const &fullname, *discover_counter_func* discover_counter, *discover_counters_mode* mode, *error_code* &ec = *throws*)

Call the supplied function for the given registered counter type.

inline *counter_status* **discover_counter_type**(*counter_info* const &info, *discover_counter_func* const &f, *discover_counters_mode* mode, *error_code* &ec = *throws*)

counter_status **get_counter_create_function**(*counter_info* const &info, *create_counter_func* &create_counter, *error_code* &ec = *throws*) const

Retrieve the counter creation function which is associated with a given counter type.

counter_status **get_counter_discovery_function**(*counter_info* const &info, *discover_counters_func* &func, *error_code* &ec) const

Retrieve the counter discovery function which is associated with a given counter type.

counter_status **remove_counter_type**(*counter_info* const &info, *error_code* &ec = *throws*)

Remove an existing counter type from the (local) registry.

> **Note:** This doesn't remove existing counters of this type, it just inhibits defining new counters using this type.

counter_status **create_raw_counter_value**(*counter_info* const &info, *std*::int64_t *countervalue, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new performance counter instance of type raw_counter based on given counter value.

counter_status **create_raw_counter**(*counter_info* const &info, *hpx*::*function*<*std*::int64_t()> const &f, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

counter_status **create_raw_counter**(*counter_info* const &info, *hpx*::*function*<*std*::int64_t(bool)> const &f, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

counter_status **create_raw_counter**(*counter_info* const &info, *hpx*::*function*<*std*::vector<*std*::int64_t>()> const &f, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

counter_status **create_raw_counter**(*counter_info* const &info, *hpx*::*function*<*std*::vector<*std*::int64_t>(bool)> const &f, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new performance counter instance of type raw_counter based on given function returning the counter value.

counter_status **create_counter**(*counter_info* const &info, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new performance counter instance based on given counter info.

counter_status **create_statistics_counter**(*counter_info* const &info, *std*::string const &base_counter_name, *std*::vector<*std*::size_t> const &parameters, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new statistics performance counter instance based on given base counter name and given base time interval (milliseconds).

counter_status **create_arithmetics_counter**(*counter_info* const &info, *std*::vector<*std*::string> const &base_counter_names, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new arithmetics performance counter instance based on given base counter names.

counter_status **create_arithmetics_counter_extended**(*counter_info* const &info, *std*::vector<*std*::string> const &base_counter_names, *naming*::gid_type &id, *error_code* &ec = *throws*)

Create a new extended arithmetics performance counter instance based on given base counter names.

counter_status **add_counter**(*hpx*::id_type const &id, *counter_info* const &info, *error_code* &ec = *throws*)

Add an existing performance counter instance to the registry.

counter_status **remove_counter**(*counter_info* const &info, *hpx*::id_type const &id, *error_code* &ec = *throws*)

remove the existing performance counter from the registry

counter_status **get_counter_type**(*std*::string const &name, *counter_info* &info, *error_code* &ec = *throws*)

Retrieve counter type information for given counter name.

Public Static Functions

static *registry* &**instance**()

Protected Functions

counter_type_map_type::iterator **locate_counter_type**(*std*::string const &type_name)

counter_type_map_type::const_iterator **locate_counter_type**(*std*::string const &type_name) const

Private Types

using **counter_type_map_type** = *std*::map<*std*::string, *counter_data*>

Private Members

counter_type_map_type **countertypes_**

struct **counter_data**

Public Functions

inline **counter_data**(*counter_info* const &info, *create_counter_func* const &create_counter, *discover_counters_func* const &discover_counters)

Public Members

counter_info **info_**

create_counter_func **create_counter_**

discover_counters_func **discover_counters_**

plugin_factories

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/plugin_factories/binary_filter_factory.hpp

Defined in header hpx/plugin_factories/binary_filter_factory.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_BINARY_FILTER_FACTORY(BinaryFilter, pluginname)

This macro is used create and to register a minimal component factory with Hpx.Plugin.

namespace **hpx**

namespace **plugins**

template<typename **BinaryFilter**>

struct **binary_filter_factory** : public binary_filter_factory_base

#include <binary_filter_factory.hpp> The *message_handler_factory* provides a minimal implementation of a message handler's factory. If no additional functionality is required this type can be used to implement the full set of minimally required functions to be exposed by a message handler's factory instance.

Template Parameters **BinaryFilter** – The message handler type this factory should be responsible for.

Public Functions

inline **binary_filter_factory**(*util*::section const *global, *util*::section const *local, bool isenabled)

Construct a new factory instance.

Note: The contents of both sections has to be cloned in order to save the configuration setting for later use.

Parameters

- **global** – [in] The pointer to a *hpx::util::section* instance referencing the settings read from the [settings] section of the global configuration file (hpx.ini) This pointer may be nullptr if no such section has been found.
- **local** – [in] The pointer to a *hpx::util::section* instance referencing the settings read from the section describing this component type: [hpx.components.<name>], where <name> is the instance name of the component as given in the configuration files.
- **isenabled** –

~binary_filter_factory() override = default

inline *serialization*::binary_filter ***create**(bool compress, *serialization*::binary_filter *next_filter = nullptr) override

Create a new instance of a message handler

return Returns the newly created instance of the message handler supported by this factory

Protected Attributes

util::section **global_settings_**

util::section **local_settings_**

bool **isenabled_**

hpx/plugin_factories/message_handler_factory.hpp

Defined in header hpx/plugin_factories/message_handler_factory.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/plugin_factories/parcelport_factory.hpp

Defined in header hpx/plugin_factories/parcelport_factory.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/plugin_factories/plugin_registry.hpp

Defined in header hpx/plugin_factories/plugin_registry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_PLUGIN_REGISTRY(...)
This macro is used create and to register a minimal plugin registry with Hpx.Plugin.

HPX_REGISTER_PLUGIN_REGISTRY_(...)

HPX_REGISTER_PLUGIN_REGISTRY_2(PluginType, pluginname)

HPX_REGISTER_PLUGIN_REGISTRY_4(PluginType, pluginname, pluginsection, pluginsuffix)

HPX_REGISTER_PLUGIN_REGISTRY_5(PluginType, pluginname, pluginstring, pluginsection, pluginsuffix)

namespace **hpx**

namespace **plugins**

template<typename **Plugin**, char const *const **Name**, char const *const **Section**, char const *const **Suffix**>
struct **plugin_registry** : public *plugin_registry_base*

#include <plugin_registry.hpp> The plugin_registry provides a minimal implementation of a plugin's registry. If no additional functionality is required this type can be used to implement the full set of minimally required functions to be exposed by a plugin's registry instance.

Template Parameters Plugin – The plugin type this registry should be responsible for.

Public Functions

inline bool **get_plugin_info**(*std*::vector<*std*::string> &fillini) override

Return the ini-information for all contained components.

Parameters fillini – [in] The module is expected to fill this vector with the ini-information (one line per vector element) for all components implemented in this module.

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

runtime_components

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/runtime_components/component_factory.hpp

Defined in header hpx/runtime_components/component_factory.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_COMPONENT(type, name, mode)

Define a component factory for a component type.

This macro is used create and to register a minimal component factory for a component type which allows it to be remotely created using the *hpx::new_<>* function.

This macro can be invoked with one, two or three arguments

Parameters

- **type** – The *type* parameter is a (fully decorated) type of the component type for which a factory should be defined.
- **name** – The *name* parameter specifies the name to use to register the factory. This should uniquely (system-wide) identify the component type. The *name* parameter must conform to the C++ identifier rules (without any namespace). If this parameter is not given, the first parameter is used.
- **mode** – The *mode* parameter has to be one of the defined enumeration values of the enumeration *hpx::components::factory_state*. The default for this parameter is *hpx::components::factory_state::enabled*.

hpx/runtime_components/component_registry.hpp

Defined in header hpx/runtime_components/component_registry.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY(...)

This macro is used create and to register a minimal component registry with Hpx.Plugin.

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_(...)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_2(ComponentType, componentname)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_3(ComponentType, componentname, state)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC(...)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_(...)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_2(ComponentType, componentname)

HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_3(ComponentType, componentname, state)

namespace **hpx**

namespace **components**

template<typename **Component**, *factory_state* **state**>

struct **component_registry** : public *component_registry_base*

#include <component_registry.hpp> The component_registry provides a minimal implementation of a component's registry. If no additional functionality is required this type can be used to implement the full set of minimally required functions to be exposed by a component's registry instance.

Template Parameters **Component** – The component type this registry should be responsible for.

Public Functions

inline bool **get_component_info**(*std*::vector<*std*::string> &fillini, *std*::string const &filepath, bool is_static = false) override

Return the ini-information for all contained components.

Parameters

- **fillini** – [in] The module is expected to fill this vector with the ini-information (one line per vector element) for all components implemented in this module.
- **filepath** –
- **is_static** –

Returns Returns *true* if the parameter *fillini* has been successfully initialized with the registry data of all implemented in this module.

inline void **register_component_type**() override

Return the unique identifier of the component type this factory is responsible for.

Parameters

- **locality** – [in] The id of the locality this factory is responsible for.
- **agas_client** – [in] The AGAS client to use for component id registration (if needed).

Returns Returns the unique identifier of the component type this factory instance is responsible for. This function throws on any error.

hpx/runtime_components/components_fwd.hpp

Defined in header hpx/runtime_components/components_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

components::*server*::*runtime_support* ***get_runtime_support_ptr**()

namespace **components**

template<typename **Component**>

struct **component_factory**

namespace **server**

namespace **stubs**

namespace **components**

hpx/runtime_components/derived_component_factory.hpp

Defined in header hpx/runtime_components/derived_component_factory.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

Defines

HPX_REGISTER_DERIVED_COMPONENT_FACTORY(...)

This macro is used create and to register a minimal component factory with Hpx.Plugin. This macro may be used if the registered component factory is the only factory to be exposed from a particular module. If more than one factory needs to be exposed the *HPX_REGISTER_COMPONENT_FACTORY* and *HPX_REGISTER_COMPONENT_MODULE* macros should be used instead.

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_(...)

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_3(ComponentType, componentname, basecomponentname)

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_4(ComponentType, componentname, basecomponentname, state)

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC(...)

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_(...)

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_3(ComponentType, componentname, basecomponentname)

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_4(ComponentType, componentname, basecomponentname, state)

hpx/runtime_components/new.hpp

Defined in header hpx/runtime_components/new.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

template<typename Component, typename... Ts> < unspecified > new_ (id_type const &locality, Ts &&... vs)

Create one or more new instances of the given Component type on the specified locality.

This function creates one or more new instances of the given Component type on the specified locality and returns a future object for the global address which can be used to reference the new component instance.

Note: This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
   hpx::new_<some_component>(hpx::find_here(), ...);
hpx::id_type id = f.get();
```

Parameters

- **locality** – [in] The global address of the locality where the new instance should be created on.
- **vs** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

Returns The function returns different types depending on its use:

- If the explicit template argument *Component* represents a component type (`traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which can be used to retrieve the global address of the newly created component.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will return a new instance of that type which can be used to refer to the newly created component instance.

template<typename Component, typename... Ts> < unspecified > local_new (Ts &&... vs)

Create one new instance of the given Component type on the current locality.

This function creates one new instance of the given Component type on the current locality and returns a future object for the global address which can be used to reference the new component instance.

Note: This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
   hpx::local_new<some_component>(...);
hpx::id_type id = f.get();
```

Note: The difference of this function to *hpx::new_* is that it can be used in cases where the supplied arguments are non-copyable and non-movable. All operations are guaranteed to be local only.

Parameters **vs** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

Returns The function returns different types depending on its use:

- If the explicit template argument *Component* represents a component type (`traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which can be used to retrieve the global address of the newly created component. If the first argument is `hpx::launch::sync` the function will directly return an `hpx::id_type`.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will return a new instance of that type which can be used to refer to the newly created component instance.

`template<typename Component, typename... Ts> < unspecified > new_ (id_type const &locality, std::size_t count, Ts &&... vs)`

Create multiple new instances of the given Component type on the specified locality.

This function creates multiple new instances of the given Component type on the specified locality and returns a future object for the global address which can be used to reference the new component instance.

Note: This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type> > f =
   hpx::new_<some_component[]>(hpx::find_here(), 10, ...);
hpx::id_type id = f.get();
```

Parameters

- **locality** – [in] The global address of the locality where the new instance should be created on.
- **count** – [in] The number of component instances to create

- **vs** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

Returns The function returns different types depending on its use:

- If the explicit template argument *Component* represents an array of a component type (i.e. *Component*[], where `traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which holds a std::vector<hpx::id_type>, where each of the items in this vector is a global address of one of the newly created components.
- If the explicit template argument *Component* represents an array of a client side object type (i.e. *Component*[], where `traits::is_client<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which holds a std::vector<hpx::id_type>, where each of the items in this vector is a client side instance of the given type, each representing one of the newly created components.

template<typename Component, typename DistPolicy, typename... Ts> < unspecified > new_ (DistPolicy const &policy, Ts &&... vs)

Create one or more new instances of the given Component type based on the given distribution policy.

This function creates one or more new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for global address which can be used to reference the new component instance(s).

Note: This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
   hpx::new_<some_component>(hpx::default_layout, ...);
hpx::id_type id = f.get();
```

Parameters

- **policy** – [in] The distribution policy used to decide where to place the newly created.
- **vs** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

Returns The function returns different types depending on its use:

- If the explicit template argument *Component* represents a component type (`traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which can be used to retrieve the global address of the newly created component.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will return a new instance of that type which can be used to refer to the newly created component instance.

template<typename Component, typename DistPolicy, typename... Ts> < unspecified > new_ (DistPolicy const &policy, std::size_t count, Ts &&... vs)

Create multiple new instances of the given Component type on the localities as defined by the given distribution policy.

This function creates multiple new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for the global address which can be used to reference the new component instance.

Note: This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type> > f =
   hpx::new_<some_component[]>(hpx::default_layout, 10, ...);
hpx::id_type id = f.get();
```

Parameters

- **policy** – [in] The distribution policy used to decide where to place the newly created.
- **count** – [in] The number of component instances to create
- **vs** – [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

Returns The function returns different types depending on its use:

- If the explicit template argument *Component* represents an array of a component type (i.e. *Component*[], where `traits::is_component<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which holds a std::vector<hpx::id_type>, where each of the items in this vector is a global address of one of the newly created components.
- If the explicit template argument *Component* represents an array of a client side object type (i.e. *Component*[], where `traits::is_client<Component>::value` evaluates to true), the function will return an *hpx::future* object instance which holds a std::vector<hpx::id_type>, where each of the items in this vector is a client side instance of the given type, each representing one of the newly created components.

runtime_distributed

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/runtime_distributed.hpp

Defined in header hpx/runtime_distributed.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

class **runtime_distributed** : public *runtime*

#include <runtime_distributed.hpp> The *runtime* class encapsulates the HPX runtime system in a simple to use way. It makes sure all required parts of the HPX runtime system are properly initialized.

Public Functions

explicit **runtime_distributed**(*util*::runtime_configuration &rtcfg, int (*pre_main)(*runtime_mode*) = nullptr, void (*post_main)() = nullptr)

Construct a new HPX runtime instance

Parameters **locality_mode** – [in] This is the mode the given runtime instance should be executed in.

~runtime_distributed()

The destructor makes sure all HPX runtime services are properly shut down before exiting.

int **start**(*hpx*::*function*<hpx_main_function_type> const &func, bool blocking = false) override

Start the runtime system.

Parameters

- **func** – [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the typedef *hpx_main_function_type*.
- **blocking** – [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function *runtime::start* will call *runtime::wait* internally.

Returns If a blocking is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter `func`. Otherwise it will return zero.

int **start**(bool blocking = false) override

Start the runtime system.

Parameters **blocking** – [in] This allows to control whether this call blocks until the runtime system has been stopped. If this parameter is *true* the function *runtime::start* will call *runtime::wait* internally .

Returns If a blocking is a true, this function will return the value as returned as the result of the invocation of the function object given by the parameter `func`. Otherwise it will return zero.

int **wait**() override

Wait for the shutdown action to be executed.

Returns This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

void **stop**(bool blocking = true) override

Initiate termination of the runtime system.

Parameters **blocking** – [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

int **finalize**(double shutdown_timeout) override

void **stop_helper**(bool blocking, *std*::condition_variable &cond, *std*::mutex &mtx)

Stop the runtime system, wait for termination.

Parameters **blocking** – [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

int **suspend**() override

Suspend the runtime system.

int **resume**() override

Resume the runtime system.

bool **report_error**(*std*::size_t num_thread, *std*::exception_ptr const &e, bool terminate_all = true) override

Report a non-recoverable error to the runtime system.

Parameters

- **num_thread** – [in] The number of the operating system thread the error has been detected in.
- **e** – [in] This is an instance encapsulating an exception which lead to this function call.
- **terminate_all** – [in] Kill all localities attached to the currently running application (default: true)

bool **report_error**(*std*::exception_ptr const &e, bool terminate_all = true) override

Report a non-recoverable error to the runtime system.

Note: This function will retrieve the number of the current shepherd thread and forward to the report_error function above.

Parameters

- **e** – [in] This is an instance encapsulating an exception which lead to this function call.
- **terminate_all** – [in] Kill all localities attached to the currently running application (default: true)

int **run**(*hpx*::*function*<hpx_main_function_type> const &func) override

Run the HPX runtime system, use the given function for the main *thread* and block waiting for all threads to finish.

Note: The parameter *func* is optional. If no function is supplied, the runtime system will simply wait for the shutdown action without explicitly executing any main thread.

Parameters **func** – [in] This is the main function of an HPX application. It will be scheduled for execution by the thread manager as soon as the runtime has been initialized. This function is expected to expose an interface as defined by the typedef *hpx_main_function_type*. This parameter is optional and defaults to none main thread function, in which case all threads have to be scheduled explicitly.

Returns This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

int **run**() override

Run the HPX runtime system, initially use the given number of (OS) threads in the thread-manager and block waiting for all threads to finish.

Returns This function will always return 0 (zero).

bool **is_networking_enabled**() override

template<typename **F**>
inline *components*::*server*::console_error_dispatcher::sink_type **set_error_sink**(*F* &&sink)

performance_counters::*registry* &**get_counter_registry**()

Allow access to the registry counter registry instance used by the HPX runtime.

performance_counters::*registry* const &**get_counter_registry**() const

Allow access to the registry counter registry instance used by the HPX runtime.

void **register_query_counters**(*std*::shared_ptr<*util*::query_counters> const &active_counters)

void **start_active_counters**(*error_code* &ec = *throws*) const

void **stop_active_counters**(*error_code* &ec = *throws*) const

void **reset_active_counters**(*error_code* &ec = *throws*) const

void **reinit_active_counters**(bool reset = true, *error_code* &ec = *throws*) const

void **evaluate_active_counters**(bool reset = false, char const *description = nullptr, *error_code* &ec = *throws*) const

void **stop_evaluating_counters**(bool terminate = false) const

naming::resolver_client &**get_agas_client**()

Allow access to the AGAS client instance used by the HPX runtime.

hpx::*threads*::*threadmanager* &**get_thread_manager**() override

Allow access to the thread manager instance used by the HPX runtime.

applier::*applier* &**get_applier**()

Allow access to the applier instance used by the HPX runtime.

std::string **here**() const override

Returns a string of the locality endpoints (usable in debug output)

naming::address_type **get_runtime_support_lva**() const

naming::gid_type **get_next_id**(*std*::size_t count = 1)

void **init_id_pool_range**()

util::unique_id_ranges &**get_id_pool**()

void **initialize_agas**()

Initialize AGAS operation.

void **add_pre_startup_function**(*startup_function_type* f) override

Add a function to be executed inside a HPX thread before hpx_main but guaranteed to be executed before any startup function registered with *add_startup_function*.

Note: The difference to a startup function is that all pre-startup functions will be (system-wide) executed before any startup function.

Parameters f – The function 'f' will be called from inside a HPX thread before hpx_main is executed. This is very useful to setup the runtime environment of the application (install performance counters, etc.)

void **add_startup_function**(*startup_function_type* f) override

Add a function to be executed inside a HPX thread before hpx_main

Parameters f – The function 'f' will be called from inside a HPX thread before hpx_main is executed. This is very useful to setup the runtime environment of the application (install performance counters, etc.)

void **add_pre_shutdown_function**(*shutdown_function_type* f) override

Add a function to be executed inside a HPX thread during hpx::finalize, but guaranteed before any of the shutdown functions is executed.

Note: The difference to a shutdown function is that all pre-shutdown functions will be (system-wide) executed before any shutdown function.

Parameters **f** – The function 'f' will be called from inside a HPX thread while hpx::finalize is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

void **add_shutdown_function**(*shutdown_function_type* f) override

Add a function to be executed inside a HPX thread during hpx::finalize

Parameters **f** – The function 'f' will be called from inside a HPX thread while hpx::finalize is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

hpx::*util*::*io_service_pool* ***get_thread_pool**(char const *name) override

Access one of the internal thread pools (io_service instances) HPX is using to perform specific tasks. The three possible values for the argument `name` are "main_pool", "io_pool", "parcel_pool", and "timer_pool". For any other argument value the function will return zero.

bool **register_thread**(char const *name, *std*::size_t num = 0, bool service_thread = true, *error_code* &ec = *throws*) override

Register an external OS-thread with HPX.

notification_policy_type **get_notification_policy**(char const *prefix, runtime_local::os_thread_type type) override

Generate a new notification policy instance for the given thread name prefix

std::uint32_t **get_locality_id**(*error_code* &ec) const override

std::size_t **get_num_worker_threads**() const override

std::uint32_t **get_num_localities**(*hpx*::*launch*::sync_policy, *error_code* &ec) const override

std::uint32_t **get_initial_num_localities**() const override

hpx::*future*<*std*::uint32_t> **get_num_localities**() const override

std::string **get_locality_name**() const override

std::uint32_t **get_num_localities**(*hpx*::*launch*::sync_policy, *components*::component_type type, *error_code* &ec) const

hpx::*future*<*std*::uint32_t> **get_num_localities**(*components*::component_type type) const

std::uint32_t **assign_cores**(*std*::string const &locality_basename, *std*::uint32_t num_threads) override

std::uint32_t **assign_cores**() override

Public Static Functions

static void **register_counter_types**()

Install all performance counters related to this runtime instance.

Private Types

using **used_cores_map_type** = *std*::map<*std*::string, *std*::uint32_t>

Private Functions

threads::thread_result_type **run_helper**(*hpx*::*function*<*runtime*::*hpx_main_function_type*> const &func, int &result)

void **init_global_data**()

void **deinit_global_data**()

void **wait_helper**(*std*::mutex &mtx, *std*::condition_variable &cond, bool &running)

void **init_tss_helper**(char const *context, runtime_local::os_thread_type type, *std*::size_t local_thread_num, *std*::size_t global_thread_num, char const *pool_name, char const *postfix, bool service_thread)

void **deinit_tss_helper**(char const *context, *std*::size_t num) const

void **init_tss_ex**(*std*::string const &locality, char const *context, runtime_local::os_thread_type type, *std*::size_t local_thread_num, *std*::size_t global_thread_num, char const *pool_name, char const *postfix, bool service_thread, *error_code* &ec) const

Private Members

runtime_mode **mode_**

util::unique_id_ranges **id_pool_**

naming::resolver_client **agas_client_**

applier::*applier* **applier_**

used_cores_map_type **used_cores_map_**

std::unique_ptr<*components*::*server*::*runtime_support*> **runtime_support_**

std::shared_ptr<*util*::query_counters> **active_counters_** int (***pre_main_**)(*runtime_mode*)

void (***post_main_**)()

Private Static Functions

static void **default_errorsink**(*std*::string const&)

hpx/runtime_distributed/applier.hpp

Defined in header hpx/runtime_distributed/applier.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **applier**

class **applier**

#include <applier.hpp> The *applier* class is used to decide whether a particular action has to be issued on a local or a remote resource. If the target component is local a new *thread* will be created, if the target is remote a parcel will be sent.

Public Functions

HPX_NON_COPYABLE(*applier*)

applier()

void **init**(*threads*::*threadmanager* &tm)

~applier() = default

void **initialize**(*std*::uint64_t rts)

threads::*threadmanager* &**get_thread_manager**()

Access the *thread-manager* instance associated with this *applier*.

This function returns a reference to the thread manager this applier instance has been created with.

naming::gid_type const &**get_raw_locality**(*error_code* &ec = *throws*) const

Allow access to the locality of the locality this applier instance is associated with.

This function returns a reference to the locality this applier instance is associated with.

std::uint32_t **get_locality_id**(*error_code* &ec = *throws*) const

Allow access to the id of the locality this applier instance is associated with.

This function returns a reference to the id of the locality this applier instance is associated with.

bool **get_raw_remote_localities**(*std*::vector<*naming*::gid_type> &locality_ids, *components*::component_type type = to_int(*hpx*::*components*::*component_enum_type*::*invalid*), *error_code* &ec = *throws*) const

Return list of localities of all remote localities registered with the AGAS service for a specific component type.

This function returns a list of all remote localities (all localities known to AGAS except the local one) supporting the given component type.

Parameters

- **locality_ids** – [out] The reference to a vector of id_types filled by the function.
- **type** – [in] The type of the component which needs to exist on the returned localities.

Returns The function returns *true* if there is at least one remote locality known to the AGAS service (!prefixes.empty()).

bool **get_remote_localities**(*std*::vector<*hpx*::id_type> &locality_ids, *components*::component_type type = to_int(*hpx*::*components*::*component_enum_type*::*invalid*), *error_code* &ec = *throws*) const

bool **get_raw_localities**(*std*::vector<*naming*::gid_type> &locality_ids, *components*::component_type type = to_int(*hpx*::*components*::*component_enum_type*::*invalid*)) const

Return list of locality_ids of all localities registered with the AGAS service for a specific component type.

This function returns a list of all localities (all localities known to AGAS except the local one) supporting the given component type.

Parameters

- **locality_ids** – [out] The reference to a vector of id_types filled by the function.
- **type** – [in] The type of the component which needs to exist on the returned localities.

Returns The function returns *true* if there is at least one remote locality known to the AGAS service (!prefixes.empty()).

bool **get_localities**(*std*::vector<*hpx*::id_type> &locality_ids, *error_code* &ec = *throws*) const

bool **get_localities**(*std*::vector<*hpx*::id_type> &locality_ids, *components*::component_type type, *error_code* &ec = *throws*) const

inline *naming*::gid_type const &**get_runtime_support_raw_gid**() const

By convention the runtime_support has a gid identical to the prefix of the locality the runtime_support is responsible for

inline *hpx*::id_type const &**get_runtime_support_gid**() const

By convention the runtime_support has a gid identical to the prefix of the locality the runtime_support is responsible for

Private Members

threads::*threadmanager* ***thread_manager_**

hpx::id_type **runtime_support_id_**

hpx/runtime_distributed/applier_fwd.hpp

Defined in header hpx/runtime_distributed/applier_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **applier**

Functions

applier &**get_applier()**

The function *get_applier* returns a reference to the (thread specific) applier instance.

applier ***get_applier_ptr()**

The function *get_applier* returns a pointer to the (thread specific) applier instance. The returned pointer is NULL if the current thread is not known to HPX or if the runtime system is not active.

namespace **applier**

The namespace *applier* contains all definitions needed for the class hpx::applier::applier and its related functionality. This namespace is part of the HPX core module.

hpx/runtime_distributed/copy_component.hpp

Defined in header hpx/runtime_distributed/copy_component.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

Functions

template<typename **Component**>
future<*hpx*::id_type> **copy**(*hpx*::id_type const &to_copy)

Copy given component to the specified target locality.

The function *copy<Component>* will create a copy of the component referenced by *to_copy* on the locality specified with *target_locality*. It returns a future referring to the newly created component instance.

Note: The new component instance is created on the locality of the component instance which is to be copied.

Parameters **to_copy** – [in] The global id of the component to copy
Template Parameters **The** – only template argument specifies the component type to create.
Returns A future representing the global id of the newly (copied) component instance.

template<typename **Component**>
future<*hpx*::id_type> **copy**(*hpx*::id_type const &to_copy, *hpx*::id_type const &target_locality)

Copy given component to the specified target locality.

The function *copy<Component>* will create a copy of the component referenced by *to_copy* on the locality specified with *target_locality*. It returns a future referring to the newly created component instance.

Parameters
- **to_copy** – [in] The global id of the component to copy
- **target_locality** – [in] The locality where the copy should be created.

Template Parameters **The** – only template argument specifies the component type to create.
Returns A future representing the global id of the newly (copied) component instance.

template<typename **Derived**, typename **Stub**, typename **Data**>
Derived **copy**(client_base<*Derived*, *Stub*, *Data*> const &to_copy, *hpx*::id_type const &target_locality = *hpx*::invalid_id)

Copy given component to the specified target locality.

The function *copy* will create a copy of the component referenced by the client side object *to_copy* on the locality specified with *target_locality*. It returns a new client side object future referring to the newly created component instance.

Note: If the second argument is omitted (or is invalid_id) the new component instance is created on the locality of the component instance which is to be copied.

Parameters
- **to_copy** – [in] The client side object representing the component to copy
- **target_locality** – [in, optional] The locality where the copy should be created (default is same locality as source).

Template Parameters **The** – only template argument specifies the component type to create.
Returns A future representing the global id of the newly (copied) component instance.

hpx::find_root_locality, hpx::find_all_localities, hpx::find_remote_localities

Defined in header hpx/runtime.hpp[786].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

hpx::id_type **find_root_locality**(*error_code* &ec = *throws*)

Return the global id representing the root locality.

The function find_root_locality() can be used to retrieve the global id usable to refer to the root locality. The root locality is the locality where the main AGAS service is hosted.

See also:

hpx::find_all_localities(), hpx::find_locality()

Note: Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will return meaningful results only if called from an HPX-thread. It will return *hpx::invalid_id* otherwise.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global id representing the root locality for this application.

std::vector<*hpx*::id_type> **find_all_localities**(*error_code* &ec = *throws*)

Return the list of global ids representing all localities available to this application.

The function find_all_localities() can be used to retrieve the global ids of all localities currently available to this application.

See also:

hpx::find_here(), hpx::find_locality()

Note: Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

[786] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

---

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global ids representing the localities currently available to this application.

std::vector<*hpx*::id_type> **find_remote_localities**(*error_code* &ec = *throws*)

Return the list of locality ids of remote localities supporting the given component type. By default this function will return the list of all remote localities (all but the current locality).

The function find_remote_localities() can be used to retrieve the global ids of all remote localities currently available to this application (i.e. all localities except the current one).

See also:

hpx::find_here(), hpx::find_locality()

Note: Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global ids representing the remote localities currently available to this application.

hpx/runtime_distributed/find_here.hpp

Defined in header hpx/runtime_distributed/find_here.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

hpx::id_type **find_here**(*error_code* &ec = *throws*)

Return the global id representing this locality.

The function find_here() can be used to retrieve the global id usable to refer to the current locality.

See also:

hpx::find_all_localities(), hpx::find_locality()

Note: Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will return meaningful results only if called from an HPX-thread. It will return *hpx::invalid_id* otherwise.

Parameters **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global id representing the locality this function has been called on.

hpx::find_locality

Defined in header hpx/runtime.hpp[787].

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

std::vector<*hpx*::id_type> **find_all_localities**(*components*::component_type type, *error_code* &ec = *throws*)

Return the list of global ids representing all localities available to this application which support the given component type.

The function find_all_localities() can be used to retrieve the global ids of all localities currently available to this application which support the creation of instances of the given component type.

See also:

hpx::find_here(), hpx::find_locality()

[787] http://github.com/STEllAR-GROUP/hpx/blob/4444c45f3668f6ff40f9561db17638415bce4838/libs/full/include/include/hpx/runtime.hpp

Note: Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

Parameters

- **type** – [in] The type of the components for which the function should return the available localities.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global ids representing the localities currently available to this application which support the creation of instances of the given component type. If no localities supporting the given component type are currently available, this function will return an empty vector.

std::vector<*hpx*::id_type> **find_remote_localities**(*components*::component_type type, *error_code* &ec = *throws*)

Return the list of locality ids of remote localities supporting the given component type. By default this function will return the list of all remote localities (all but the current locality).

The function find_remote_localities() can be used to retrieve the global ids of all remote localities currently available to this application (i.e. all localities except the current one) which support the creation of instances of the given component type.

See also:

hpx::find_here(), hpx::find_locality()

Note: Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

Parameters

- **type** – [in] The type of the components for which the function should return the available remote localities.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global ids representing the remote localities currently available to this application.

hpx::id_type **find_locality**(*components*::component_type type, *error_code* &ec = *throws*)

Return the global id representing an arbitrary locality which supports the given component type.

The function find_locality() can be used to retrieve the global id of an arbitrary locality currently available to this application which supports the creation of instances of the given component type.

See also:

hpx::find_here(), hpx::find_all_localities()

Note: Generally, the id of a locality can be used for instance to create new instances of components and to invoke plain actions (global functions).

Note: As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Note: This function will return meaningful results only if called from an HPX-thread. It will return *hpx::invalid_id* otherwise.

Parameters

- **type** – [in] The type of the components for which the function should return any available locality.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Returns The global id representing an arbitrary locality currently available to this application which supports the creation of instances of the given component type. If no locality supporting the given component type is currently available, this function will return *hpx::invalid_id*.

hpx/runtime_distributed/get_locality_name.hpp

Defined in header hpx/runtime_distributed/get_locality_name.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

future<*std*::string> **get_locality_name**(*hpx*::id_type const &id)

Return the name of the referenced locality.

This function returns a future referring to the name for the locality of the given id.

See also:

std::string get_locality_name()

Parameters id – [in] The global id of the locality for which the name should be retrieved

Returns This function returns the name for the locality of the given id. The name is retrieved from the underlying networking layer and may be different for different parcel ports.

hpx/runtime_distributed/get_num_localities.hpp

Defined in header hpx/runtime_distributed/get_num_localities.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

Functions

hpx::*future*<*std*::uint32_t> **get_num_localities**(*components*::component_type t)

Asynchronously return the number of localities which are currently registered for the running application.

The function *get_num_localities* asynchronously returns the number of localities currently connected to the console which support the creation of the given component type. The returned future represents the actual result.

See also:

hpx::find_all_localities, *hpx::get_num_localities*

Note: This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

Parameters t – The component type for which the number of connected localities should be retrieved.

std::uint32_t **get_num_localities**(*launch*::sync_policy, *components*::component_type t, *error_code* &ec = *throws*)

Synchronously return the number of localities which are currently registered for the running application.

The function *get_num_localities* returns the number of localities currently connected to the console which support the creation of the given component type. The returned future represents the actual result.

See also:

hpx::find_all_localities, *hpx::get_num_localities*

Note: This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

Parameters

- **t** – The component type for which the number of connected localities should be retrieved.
- **ec** – [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

hpx/runtime_distributed/migrate_component.hpp

Defined in header hpx/runtime_distributed/migrate_component.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

Functions

template<typename **Component**, typename **DistPolicy**>
future<*hpx*::id_type> **migrate**(*hpx*::id_type const &to_migrate, [[maybe_unused]] *DistPolicy* const &policy)

Migrate the given component to the specified target locality

The function *migrate<Component>* will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Parameters

- **to_migrate** – [in] The client side representation of the component to migrate.
- **policy** – [in] A distribution policy which will be used to determine the locality to migrate this object to.

Template Parameters

- **Component** – Specifies the component type of the component to migrate.
- **DistPolicy** – Specifies the distribution policy to use to determine the destination locality.

Returns A future representing the global id of the migrated component instance. This should be the same as *migrate_to*.

template<typename **Derived**, typename **Stub**, typename **Data**, typename **DistPolicy**>
Derived **migrate**(client_base<*Derived*, *Stub*, *Data*> const &to_migrate, *DistPolicy* const &policy)

Migrate the given component to the specified target locality

The function *migrate<Component>* will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Parameters

- **to_migrate** – [in] The client side representation of the component to migrate.

- **policy** – [in] A distribution policy which will be used to determine the locality to migrate this object to.

Template Parameters
- **Derived** – Specifies the component type of the component to migrate.
- **DistPolicy** – Specifies the distribution policy to use to determine the destination locality.

Returns A future representing the global id of the migrated component instance. This should be the same as *migrate_to*.

template<typename **Component**>
future<*hpx*::id_type> **migrate**(*hpx*::id_type const &to_migrate, *hpx*::id_type const &target_locality)

Migrate the component with the given id to the specified target locality

The function *migrate<Component>* will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Parameters
- **to_migrate** – [in] The global id of the component to migrate.
- **target_locality** – [in] The locality where the component should be migrated to.

Template Parameters **Component** – Specifies the component type of the component to migrate.

Returns A future representing the global id of the migrated component instance. This should be the same as *migrate_to*.

template<typename **Derived**, typename **Stub**, typename **Data**>
Derived **migrate**(client_base<*Derived*, *Stub*, *Data*> const &to_migrate, *hpx*::id_type const &target_locality)

Migrate the given component to the specified target locality

The function *migrate<Component>* will migrate the component referenced by *to_migrate* to the locality specified with *target_locality*. It returns a future referring to the migrated component instance.

Parameters
- **to_migrate** – [in] The client side representation of the component to migrate.
- **target_locality** – [in] The id of the locality to migrate this object to.

Template Parameters **Derived** – Specifies the component type of the component to migrate.

Returns A client side representation of representing of the migrated component instance. This should be the same as *migrate_to*.

hpx/runtime_distributed/runtime_fwd.hpp

Defined in header hpx/runtime_distributed/runtime_fwd.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/runtime_distributed/runtime_support.hpp

Defined in header hpx/runtime_distributed/runtime_support.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **agas**

Functions

struct runtime_components_init_interface_functions &**runtime_components_init**()

namespace **components**

Functions

struct counter_interface_functions &**counter_init**()

class **runtime_support** : public *hpx*::*components*::*stubs*::*runtime_support*

#include <runtime_support.hpp> The runtime_support class is the client side representation of a server::runtime_support component

Public Functions

inline **runtime_support**(*hpx*::id_type const &gid = *hpx*::invalid_id)

Create a client side representation for the existing server::runtime_support instance with the given global id *gid*.

template<typename **Component**, typename ...**Ts**>
inline *hpx*::id_type **create_component**(*Ts*&&... vs)

Create a new component type using the *runtime_support*.

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*hpx*::id_type> **create_component_async**(*Ts*&&... vs)

Asynchronously create a new component using the *runtime_support*.

template<typename **Component**, typename ...**Ts**>
inline *std*::vector<*hpx*::id_type> **bulk_create_component**(*std*::size_t, *Ts*&&... vs)

Asynchronously create N new default constructed components using the *runtime_support*

template<typename **Component**, typename ...**Ts**>
inline *hpx*::*future*<*std*::vector<*hpx*::id_type>> **bulk_create_components_async**(*std*::size_t, *Ts*&&... vs)

Asynchronously create a new component using the *runtime_support*.

inline *hpx*::*future*<int> **load_components_async**()

inline int **load_components**()

inline *hpx*::*future*<void> **call_startup_functions_async**(bool pre_startup)

inline void **call_startup_functions**(bool pre_startup)

inline *hpx*::*future*<void> **shutdown_async**(double timeout = -1)

Shutdown the given runtime system.

inline void **shutdown**(double timeout = -1)

inline void **shutdown_all**(double timeout = -1)

Shutdown the runtime systems of all localities.

inline *hpx*::*future*<void> **terminate_async**()

Terminate the given runtime system.

inline void **terminate**()

inline void **terminate_all**()

Terminate the runtime systems of all localities.

inline void **get_config**(*util*::section &ini)

Retrieve configuration information.

inline *hpx*::id_type const &**get_id**() const

inline *naming*::gid_type const &**get_raw_gid**() const

Private Types

typedef *stubs*::*runtime_support* **base_type**

Private Members

hpx::id_type **gid_**

hpx/runtime_distributed/server/copy_component.hpp

Defined in header hpx/runtime_distributed/server/copy_component.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

namespace **server**

Functions

template<typename **Component**>
future<*hpx*::id_type> **copy_component_here**(*hpx*::id_type const &to_copy)

template<typename **Component**>
future<*hpx*::id_type> **copy_component**(*hpx*::id_type const &to_copy, *hpx*::id_type const &target_locality)

template<typename **Component**>

struct **copy_component_action** : public *hpx*::*actions*::action<*future*<*hpx*::id_type>(*)(*hpx*::id_type const&, *hpx*::id_type const&), &*copy_component*<*Component*>, *copy_component_action*<*Component*>>

template<typename **Component**>

struct **copy_component_action_here** : public *hpx*::*actions*::action<*future*<*hpx*::id_type>(*)(*hpx*::id_type const&), &*copy_component_here*<*Component*>, *copy_component_action_here*<*Component*>>

hpx/runtime_distributed/server/runtime_support.hpp

Defined in header hpx/runtime_distributed/server/runtime_support.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

namespace **server**

class **runtime_support**

Public Types

typedef *runtime_support* **type_holder**

Public Functions

explicit **runtime_support**(*hpx*::*util*::runtime_configuration &cfg)

inline **~runtime_support**()

void **delete_function_lists**()

void **tidy**()

template<typename **Component**>
naming::gid_type **create_component**()

Actions to create new objects.

template<typename **Component**, typename **T**, typename ...**Ts**>
naming::gid_type **create_component**(*T* v, *Ts*... vs)

template<typename **Component**>
std::vector<*naming*::gid_type> **bulk_create_component**(*std*::size_t count)

template<typename **Component**, typename **T**, typename ...**Ts**>

std::vector<*naming*::gid_type> **bulk_create_component**(*std*::size_t count, *T* v, *Ts*... vs)

template<typename **Component**>
naming::gid_type **copy_create_component**(*std*::shared_ptr<*Component*> const &p, bool)

template<typename **Component**>
naming::gid_type **migrate_component_to_here**(*std*::shared_ptr<*Component*> const &p, *hpx*::id_type)

void **shutdown**(double timeout, *hpx*::id_type const &respond_to)

Gracefully shutdown this runtime system instance.

void **shutdown_all**(double timeout)

Gracefully shutdown runtime system instances on all localities.

void **terminate**(*hpx*::id_type const &respond_to)

Shutdown this runtime system instance.

inline void **terminate_act**(*hpx*::id_type const &id)

void **terminate_all**()

Shutdown runtime system instances on all localities.

inline void **terminate_all_act**()

util::section **get_config**()

Retrieve configuration information.

int **load_components**()

Load all components on this locality.

void **call_startup_functions**(bool pre_startup)

void **call_shutdown_functions**(bool pre_shutdown)

void **garbage_collect**()

Force a garbage collection operation in the AGAS layer.

naming::gid_type **create_performance_counter**(*performance_counters*::*counter_info* const &info)

Create the given performance counter instance.

void **remove_from_connection_cache**(*naming*::gid_type const &gid, *parcelset*::endpoints_type const &eps)

Remove the given locality from our connection cache.

HPX_DEFINE_COMPONENT_ACTION (runtime_support, terminate_act, terminate_action) HPX_DEFINE_COMPONENT_ACTION(runtime_support

termination detection

terminate_all_action HPX_DEFINE_COMPONENT_ACTION (runtime_support, remove_from_connection_cache) void run()

Start the *runtime_support* component.

void **wait**()

Wait for the *runtime_support* component to notify the calling thread.

This function will be called from the main thread, causing it to block while the HPX functionality is executed. The main thread will block until the shutdown_action is executed, which in turn notifies all waiting threads.

void **stop**(double timeout, *hpx*::id_type const &respond_to, bool remove_from_remote_caches)

Notify all waiting (blocking) threads allowing the system to be properly stopped.

Note: This function can be called from any thread.

void **stopped**()

called locally only

void **notify_waiting_main**()

inline bool **was_stopped**() const

void **add_pre_startup_function**(*startup_function_type* f)

void **add_startup_function**(*startup_function_type* f)

void **add_pre_shutdown_function**(*shutdown_function_type* f)

void **add_shutdown_function**(*shutdown_function_type* f)

void **remove_here_from_connection_cache**()

void **remove_here_from_console_connection_cache**()

Public Members

terminate_all_act

Public Static Functions

static inline component_type **get_component_type**()

static inline void **set_component_type**(component_type t)

static inline constexpr void **finalize**()

finalize() will be called just before the instance gets destructed

Parameters

- **self** – [in] The HPX *thread* used to execute this function.
- **appl** – [in] The applier to be used for finalization of the component instance.

static inline bool **is_target_valid**(*hpx*::id_type const &id)

Protected Functions

int **load_components**(*util*::section &ini, *naming*::gid_type const &prefix, *naming*::resolver_client &agas_client, *hpx*::program_options::options_description &options, *std*::set<*std*::string> &startup_handled)

bool **load_component**(*hpx*::*util*::plugin::dll &d, *util*::section &ini, *std*::string const &instance, *std*::string const &component, *filesystem*::path const &lib, *naming*::gid_type const &prefix, *naming*::resolver_client &agas_client, bool isdefault, bool isenabled, *hpx*::program_options::options_description &options, *std*::set<*std*::string> &startup_handled)

bool **load_component_dynamic**(*util*::section &ini, *std*::string const &instance, *std*::string const &component, *filesystem*::path lib, *naming*::gid_type const &prefix, *naming*::resolver_client &agas_client, bool isdefault, bool isenabled, *hpx*::program_options::options_description &options, *std*::set<*std*::string> &startup_handled)

bool **load_startup_shutdown_functions**(*hpx*::*util*::plugin::dll &d, *error_code* &ec)

bool **load_commandline_options**(*hpx*::*util*::plugin::dll &d, *hpx*::program_options::options_description &options, *error_code* &ec)

bool **load_component_static**(*util*::section &ini, *std*::string const &instance, *std*::string const &component, *filesystem*::path const &lib, *naming*::gid_type const &prefix, *naming*::resolver_client &agas_client, bool isdefault, bool isenabled, *hpx*::program_options::options_description &options, *std*::set<*std*::string> &startup_handled)

bool **load_startup_shutdown_functions_static**(*std*::string const &mod, *error_code* &ec)

bool **load_commandline_options_static**(*std*::string const &mod, *hpx*::program_options::options_description &options, *error_code* &ec)

bool **load_plugins**(*util*::section &ini, *hpx*::program_options::options_description &options, *std*::set<*std*::string> &startup_handled)

bool **load_plugin**(*hpx*::*util*::plugin::dll &d, *util*::section &ini, *std*::string const &instance, *std*::string const &component, *filesystem*::path const &lib, bool isenabled, *hpx*::program_options::options_description &options, *std*::set<*std*::string> &startup_handled)

bool **load_plugin_dynamic**(*util*::section &ini, *std*::string const &instance, *std*::string const &component, *filesystem*::path lib, bool isenabled, *hpx*::program_options::options_description &options, *std*::set<*std*::string> &startup_handled)

std::size_t **dijkstra_termination_detection**(*std*::vector<*hpx*::id_type> const &locality_ids)

Private Types

typedef *hpx*::*spinlock* **plugin_map_mutex_type**

typedef *plugin_factory* **plugin_factory_type**

typedef *std*::map<*std*::string, *plugin_factory_type*> **plugin_map_type**

typedef *std*::map<*std*::string, *hpx*::*util*::plugin::dll> **modules_map_type**

typedef *std*::vector<static_factory_load_data_type> **static_modules_type**

Private Members

std::mutex **mtx_**

std::condition_variable **wait_condition_**

std::condition_variable **stop_condition_**

bool **stop_called_**

bool **stop_done_**

bool **terminated_**

std::thread::id **main_thread_id_**

std::atomic<bool> **shutdown_all_invoked_**

plugin_map_mutex_type **p_mtx_**

plugin_map_type **plugins_**

modules_map_type &**modules_**

static_modules_type **static_modules_**

hpx::*spinlock* **globals_mtx_**

std::list<*startup_function_type*> **pre_startup_functions_** *std*::list<*startup_function_type*> **startup_functions_**

std::list<*shutdown_function_type*> **pre_shutdown_functions_**

std::list<*shutdown_function_type*> **shutdown_functions_**

struct **plugin_factory**

Public Functions

inline **plugin_factory**(*std*::shared_ptr<*plugins*::plugin_factory_base> const &f, *hpx*::*util*::plugin::dll const &d, bool enabled)

Public Members

std::shared_ptr<*plugins*::plugin_factory_base> **first**

hpx::*util*::plugin::dll const &**second**

bool **isenabled**

hpx/runtime_distributed/stubs/runtime_support.hpp

Defined in header hpx/runtime_distributed/stubs/runtime_support.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **components**

namespace **stubs**

struct **runtime_support**

Subclassed by *hpx::components::runtime_support*

Public Static Functions

template<typename **Component**, typename ...**Ts**>
static inline *hpx*::*future*<*hpx*::id_type> **create_component_async**(*hpx*::id_type const &gid, *Ts*&&... vs)

Create a new component *type* using the *runtime_support* with the given *targetgid*. This is a non-blocking call. The caller needs to call *future::get* on the result of this function to obtain the global id of the newly created object.

template<typename **Component**, typename ...**Ts**>
static inline *hpx*::id_type **create_component**(*hpx*::id_type const &gid, *Ts*&&... vs)

Create a new component *type* using the *runtime_support* with the given *targetgid*. Block for the creation to finish.

template<typename **Component**, typename ...**Ts**>
static inline *hpx*::*future*<*std*::vector<*hpx*::id_type>> **bulk_create_component_colocated_async**(*hpx*::id_type const &gid, *std*::size_t count, *Ts*&&... vs)

Create multiple new components *type* using the *runtime_support* colocated with the with the given *targetgid*. This is a non-blocking call.

template<typename **Component**, typename ...**Ts**>
static inline *std*::vector<*hpx*::id_type> **bulk_create_component_colocated**(*hpx*::id_type const &gid, *std*::size_t count, *Ts*&&... vs)

Create multiple new components *type* using the *runtime_support* colocated with the with the given *targetgid*. Block for the creation to finish.

template<typename **Component**, typename ...**Ts**>
static inline *hpx*::*future*<*std*::vector<*hpx*::id_type>> **bulk_create_component_async**(*hpx*::id_type const &gid, *std*::size_t count, *Ts*&&... vs)

Create multiple new components *type* using the *runtime_support* on the given locality. This is a non-blocking call.

template<typename **Component**, typename ...**Ts**>
static inline *std*::vector<*hpx*::id_type> **bulk_create_component**(*hpx*::id_type const &gid, *std*::size_t count, *Ts*&&... vs)

Create multiple new components *type* using the *runtime_support* on the given locality. Block for the creation to finish.

template<typename **Component**, typename ...**Ts**>
static inline *hpx*::*future*<*hpx*::id_type> **create_component_colocated_async**(*hpx*::id_type const &gid, *Ts*&&... vs)

Create a new component *type* using the *runtime_support* with the given *targetgid*. This is a non-blocking call. The caller needs to call *future::get* on the result of this function to obtain the global id of the newly created object.

template<typename **Component**, typename ...**Ts**>
static inline *hpx*::id_type **create_component_colocated**(*hpx*::id_type const &gid, *Ts*&&... vs)

Create a new component *type* using the *runtime_support* with the given *targetgid*. Block for the creation to finish.

template<typename **Component**>
static inline *hpx*::*future*<*hpx*::id_type> **copy_create_component_async**(*hpx*::id_type const &gid, *std*::shared_ptr<*Component*> const &p, bool local_op)

template<typename **Component**>
static inline *hpx*::id_type **copy_create_component**(*hpx*::id_type const &gid, *std*::shared_ptr<*Component*> const &p, bool local_op)

template<typename **Component**>
static inline *hpx*::*future*<*hpx*::id_type> **migrate_component_async**(*hpx*::id_type const &target_locality, *std*::shared_ptr<*Component*> const &p, *hpx*::id_type const &to_migrate)

template<typename **Component**, typename **DistPolicy**>
static inline *hpx*::*future*<*hpx*::id_type> **migrate_component_async**(*DistPolicy* const &policy, *std*::shared_ptr<*Component*> const &p, *hpx*::id_type const &to_migrate)

template<typename **Component**, typename **Target**>
static inline *hpx*::id_type **migrate_component**(*Target* const &target, *hpx*::id_type const &to_migrate, *std*::shared_ptr<*Component*> const &p)

static *hpx*::*future*<int> **load_components_async**(*hpx*::id_type const &gid)

static int **load_components**(*hpx*::id_type const &gid)

static *hpx*::*future*<void> **call_startup_functions_async**(*hpx*::id_type const &gid, bool pre_startup)

static void **call_startup_functions**(*hpx*::id_type const &gid, bool pre_startup)

static *hpx*::*future*<void> **shutdown_async**(*hpx*::id_type const &targetgid, double timeout = -1)

Shutdown the given runtime system.

static void **shutdown**(*hpx*::id_type const &targetgid, double timeout = -1)

static void **shutdown_all**(*hpx*::id_type const &targetgid, double timeout = -1)

Shutdown the runtime systems of all localities.

static void **shutdown_all**(double timeout = -1)

static *hpx*::*future*<void> **terminate_async**(*hpx*::id_type const &targetgid)

Retrieve configuration information.

Terminate the given runtime system

static void **terminate**(*hpx*::id_type const &targetgid)

static void **terminate_all**(*hpx*::id_type const &targetgid)

Terminate the runtime systems of all localities.

static void **terminate_all**()

static void **garbage_collect_non_blocking**(*hpx*::id_type const &targetgid)

static *hpx*::*future*<void> **garbage_collect_async**(*hpx*::id_type const &targetgid)

static void **garbage_collect**(*hpx*::id_type const &targetgid)

static *hpx*::*future*<*hpx*::id_type> **create_performance_counter_async**(*hpx*::id_type targetgid, *performance_counters*::*counter_info* const &info)

static *hpx*::id_type **create_performance_counter**(*hpx*::id_type targetgid, *performance_counters*::*counter_info* const &info, *error_code* &ec = *throws*)

static *hpx*::*future*<*util*::section> **get_config_async**(*hpx*::id_type const &targetgid)

Retrieve configuration information.

static void **get_config**(*hpx*::id_type const &targetgid, *util*::section &ini)

static void **remove_from_connection_cache_async**(*hpx*::id_type const &target, *naming*::gid_type const &gid, *parcelset*::endpoints_type const &endpoints)

segmented_algorithms

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parallel/segmented_algorithms/adjacent_difference.hpp

Defined in header hpx/parallel/segmented_algorithms/adjacent_difference.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *FwdIter2*> **tag_invoke**(*hpx*::adjacent_difference_t, *ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op)

template<typename **InIter1**, typename **InIter2**, typename **Op**>
InIter2 **tag_invoke**(*hpx*::adjacent_difference_t, *InIter1* first, *InIter1* last, *InIter2* dest, *Op* &&op)

hpx/parallel/segmented_algorithms/adjacent_find.hpp

Defined in header hpx/parallel/segmented_algorithms/adjacent_find.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **InIter**, typename **Pred**>
InIter **tag_invoke**(*hpx*::adjacent_find_t, *InIter* first, *InIter* last, *Pred* &&pred = *Pred*())

template<typename **ExPolicy**, typename **SegIter**, typename **Pred**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *SegIter*>::type **tag_invoke**(*hpx*::adjacent_find_t, *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *Pred* &&pred)

hpx/parallel/segmented_algorithms/all_any_none.hpp

Defined in header hpx/parallel/segmented_algorithms/all_any_none.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **InIter**, typename **F**>
bool **tag_invoke**(*hpx*::none_of_t, *InIter* first, *InIter* last, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, bool>::type **tag_invoke**(*hpx*::none_of_t, *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *F* &&f)

template<typename **InIter**, typename **F**>
bool **tag_invoke**(*hpx*::any_of_t, *InIter* first, *InIter* last, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, bool>::type **tag_invoke**(*hpx*::any_of_t, *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *F* &&f)

template<typename **InIter**, typename **F**>
bool **tag_invoke**(*hpx*::all_of_t, *InIter* first, *InIter* last, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, bool>::type **tag_invoke**(*hpx*::all_of_t, *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *F* &&f)

hpx/parallel/segmented_algorithms/count.hpp

Defined in header hpx/parallel/segmented_algorithms/count.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **InIter**, typename **T**>
std::iterator_traits<*InIter*>::difference_type **tag_invoke**(*hpx*::count_t, *InIter* first, *InIter* last, *T* const &value)

template<typename **ExPolicy**, typename **SegIter**, typename **T**>

hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<SegIter>::difference_type>::type **tag_i**

template<typename **InIter**, typename **F**>
std::iterator_traits<InIter>::difference_type **tag_invoke**(hpx::count_if_t, InIter first, InIter last, F &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
hpx::parallel::util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<SegIter>::difference_type>::type **tag_i**

hpx/parallel/segmented_algorithms/exclusive_scan.hpp

Defined in header hpx/parallel/segmented_algorithms/exclusive_scan.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **InIter**, typename **OutIter**, typename **T**, typename **Op** = *std*::plus<*T*>>
OutIter **tag_invoke**(*hpx*::exclusive_scan_t, *InIter* first, *InIter* last, *OutIter* dest, *T* init, *Op* &&op = *Op*())

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Op** = *std*::plus<*T*>>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **tag_invoke**(*hpx*::exclusive_scan_t, *ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *T* init, *Op* &&op = *Op*())

hpx/parallel/segmented_algorithms/fill.hpp

Defined in header hpx/parallel/segmented_algorithms/fill.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

hpx/parallel/segmented_algorithms/for_each.hpp

Defined in header hpx/parallel/segmented_algorithms/for_each.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **InIter**, typename **F**>
InIter **tag_invoke**(*hpx*::for_each_t, *InIter* first, *InIter* last, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *SegIter*>::type **tag_invoke**(*hpx*::for_each_t, *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *F* &&f)

template<typename **InIter**, typename **Size**, typename **F**>
InIter **tag_invoke**(*hpx*::for_each_n_t, *InIter* first, *Size* count, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **Size**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *SegIter*>::type **tag_invoke**(*hpx*::for_each_n_t, *ExPolicy* &&policy, *SegIter* first, *Size* count, *F* &&f)

hpx/parallel/segmented_algorithms/generate.hpp

Defined in header hpx/parallel/segmented_algorithms/generate.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **SegIter**, typename **F**>
SegIter **tag_invoke**(*hpx*::generate_t, *SegIter* first, *SegIter* last, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *SegIter*>::type **tag_invoke**(*hpx*::generate_t, *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *F* &&f)

hpx/parallel/segmented_algorithms/inclusive_scan.hpp

Defined in header hpx/parallel/segmented_algorithms/inclusive_scan.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **InIter**, typename **OutIter**, typename **Op** = *std*::plus<typename *std*::iterator_traits<*InIter*>::value_type>>
OutIter **tag_invoke**(*hpx*::inclusive_scan_t, *InIter* first, *InIter* last, *OutIter* dest, *Op* &&op = *Op*())

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op** = *std*::plus<typename *std*::iterator_traits<*FwdIter1*>::value_type>>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **tag_invoke**(*hpx*::inclusive_scan_t, *ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op = *Op*())

template<typename **InIter**, typename **OutIter**, typename **Op**, typename **T**>

OutIter **tag_invoke**(*hpx*::inclusive_scan_t, *InIter* first, *InIter* last, *OutIter* dest, *Op* &&op, *T* &&init)

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**, typename **T**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **tag_invoke**(*hpx*::inclusive_scan_t, *ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op, *T* &&init)

hpx/parallel/segmented_algorithms/minmax.hpp

Defined in header hpx/parallel/segmented_algorithms/minmax.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

Typedefs

template<typename **T**>

using **minmax_element_result** = *hpx*::*parallel*::*util*::min_max_result<*T*>

namespace **segmented**

Typedefs

template<typename **T**>

using **minmax_element_result** = *hpx*::*parallel*::*util*::min_max_result<*T*>

Functions

template<typename **SegIter**, typename **F**>
SegIter **tag_invoke**(*hpx*::min_element_t, *SegIter* first, *SegIter* last, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result_t<*ExPolicy*, *SegIter*> **tag_invoke**(*hpx*::min_element_t, *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *F* &&f)

template<typename **SegIter**, typename **F**>
SegIter **tag_invoke**(*hpx*::max_element_t, *SegIter* first, *SegIter* last, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>

hpx::parallel::util::detail::algorithm_result_t<ExPolicy, SegIter> **tag_invoke**(hpx::max_element_t, ExPolicy &&policy, SegIter first, SegIter last, F &&f)

template<typename **SegIter**, typename **F**>
minmax_element_result<SegIter> **tag_invoke**(hpx::minmax_element_t, SegIter first, SegIter last, F &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **F**>
hpx::parallel::util::detail::algorithm_result_t<ExPolicy, minmax_element_result<SegIter>> **tag_invoke**(hpx::minmax_ele Ex-Pol-icy &&pol-icy, Se-gIter first, Se-gIter last, F &&f)

hpx/parallel/segmented_algorithms/reduce.hpp

Defined in header hpx/parallel/segmented_algorithms/reduce.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **InIterB**, typename **InIterE**, typename **T**, typename **F**>
T **tag_invoke**(hpx::reduce_t, InIterB first, InIterE last, T init, F &&f)

template<typename **ExPolicy**, typename **InIterB**, typename **InIterE**, typename **T**, typename **F**>
parallel::util::detail::algorithm_result<ExPolicy, T>::type **tag_invoke**(hpx::reduce_t, ExPolicy &&policy, InIterB first, InIterE last, T init, F &&f)

hpx/parallel/segmented_algorithms/transform.hpp

Defined in header hpx/parallel/segmented_algorithms/transform.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **SegIter**, typename **OutIter**, typename **F**>
hpx::*parallel*::*util*::in_out_result<*SegIter*, *OutIter*> **tag_invoke**(*hpx*::transform_t, *SegIter* first, *SegIter* last, *OutIter* dest, *F* &&f)

template<typename **ExPolicy**, typename **SegIter**, typename **OutIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*parallel*::*util*::in_out_result<*SegIter*, *OutIter*>>::type **tag_invo**

template<typename **InIter1**, typename **InIter2**, typename **OutIter**, typename **F**>
hpx::*parallel*::*util*::in_in_out_result<*InIter1*, *InIter2*, *OutIter*> **tag_invoke**(*hpx*::transform_t, *InIter1* first1, *InIter1* last1, *InIter2* first2, *OutIter* dest, *F* &&f)

template<typename **ExPolicy**, typename **InIter1**, typename **InIter2**, typename **OutIter**, typename **F**>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*parallel*::*util*::in_in_out_result<*InIter1*, *InIter2*, *OutIter*>>::type

template<typename **InIter1**, typename **InIter2**, typename **OutIter**, typename **F**>
hpx::*parallel*::*util*::in_in_out_result<*InIter1*, *InIter2*, *OutIter*> **tag_invoke**(*hpx*::transform_t, *InIter1* first1, *InIter1* last1, *InIter2* first2, *InIter2* last2, *OutIter* dest, *F* &&f)

template<typename **ExPolicy**, typename **InIter1**, typename **InIter2**, typename **OutIter**, typename **F**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*parallel*::*util*::in_in_out_result<*InIter1*, *InIter2*, *OutIter*>>::type

hpx/parallel/segmented_algorithms/transform_exclusive_scan.hpp

Defined in header hpx/parallel/segmented_algorithms/transform_exclusive_scan.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **segmented**

Functions

template<typename **InIter**, typename **OutIter**, typename **T**, typename **Op**, typename **Conv**>
OutIter **tag_invoke**(*hpx*::transform_exclusive_scan_t, *InIter* first, *InIter* last, *OutIter* dest, *T* init, *Op* &&op, *Conv* &&conv)

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Op**, typename **Conv**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **tag_invoke**(*hpx*::transform_exclusive_scan_t, *ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *T* init, *Op* &&op, *Conv* &&conv)

hpx/parallel/segmented_algorithms/transform_inclusive_scan.hpp

Defined in header hpx/parallel/segmented_algorithms/transform_inclusive_scan.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **segmented**

Functions

template<typename **InIter**, typename **OutIter**, typename **Op**, typename **Conv**>
OutIter **tag_invoke**(*hpx*::transform_inclusive_scan_t, *InIter* first, *InIter* last, *OutIter* dest, *Op* &&op, *Conv* &&conv)

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**, typename **Conv**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **tag_invoke**(*hpx*::transform_inclusive_scan_t, *ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op, *Conv* &&conv)

template<typename **InIter**, typename **OutIter**, typename **T**, typename **Op**, typename **Conv**>

OutIter **tag_invoke**(*hpx*::transform_inclusive_scan_t, *InIter* first, *InIter* last, *OutIter* dest, *Op* &&op, *Conv* &&conv, *T* init)

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Op**, typename **Conv**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **tag_invoke**(*hpx*::transform_inclusive_scan_t, *ExPolicy* &&policy, *FwdIter1* first, *FwdIter1* last, *FwdIter2* dest, *Op* &&op, *Conv* &&conv, *T* init)

hpx/parallel/segmented_algorithms/transform_reduce.hpp

Defined in header hpx/parallel/segmented_algorithms/transform_reduce.hpp.

See *Public API* for a list of names and headers that are part of the public *HPX* API.

namespace **hpx**

namespace **parallel**

namespace **segmented**

Functions

template<typename **SegIter**, typename **T**, typename **Reduce**, typename **Convert**>
std::decay<*T*> **tag_invoke**(*hpx*::transform_reduce_t, *SegIter* first, *SegIter* last, *T* &&init, *Reduce* &&red_op, *Convert* &&conv_op)

template<typename **ExPolicy**, typename **SegIter**, typename **T**, typename **Reduce**, typename **Convert**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, typename *std*::decay<*T*>::type>::type **tag_invoke**(*hpx*::transform_reduc *ExPolicy* &&policy, *SegIter* first, *SegIter* last, *T* &&init, *Reduce* &&red_op, *Convert* &&conv_op)

template<typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Reduce**, typename **Convert**>
T **tag_invoke**(*hpx*::transform_reduce_t, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Reduce**, typename **Convert**>
parallel::*util*::detail::algorithm_result<*ExPolicy*, *T*>::type **tag_invoke**(*hpx*::transform_reduce_t, *ExPolicy* &&policy, *FwdIter1* first1, *FwdIter1* last1, *FwdIter2* first2, *T* init, *Reduce* &&red_op, *Convert* &&conv_op)

2.9 Contributing to *HPX*

HPX development happens on Github. The following sections are a collection of useful information related to *HPX* development.

2.9.1 Contributing to *HPX*

The main source of information to understand the process of how to contribute to HPX can be found in this document[788]. This is a living document that is constantly updated with relevant information.

2.9.2 *HPX* governance model

The *HPX* project is a meritocratic, consensus-based community project. Anyone with an interest in the project can join the community, contribute to the project design and participate in the decision making process. This document[789] describes how that participation takes place and how to set about earning merit within the project community.

2.9.3 Release procedure for *HPX*

Below is a step by step procedure for making an *HPX* release. We aim to produce two releases per year: one in March-April, and one in September-October.

This is a living document and may not be totally current or accurate. It is an attempt to capture current practices in making an *HPX* release. Please update it as appropriate.

One way to use this procedure is to print a copy and check off the lines as they are completed to avoid confusion.

1. Notify developers that a release is imminent.
2. For minor and major releases: create and check out a new branch at an appropriate point on `master` with the name `release-major.minor.X`. `major` and `minor` should be the major and minor versions of the release. For patch releases: check out the corresponding `release-major.minor.X` branch.
3. Write release notes in `docs/sphinx/releases/whats_new_$VERSION.rst`. Keep adding merged PRs and closed issues to this until just before the release is made. Use `tools/generate_pr_issue_list.sh` to generate the lists. Add the new release notes to the table of contents in `docs/sphinx/releases.rst`.

[788] https://github.com/STEllAR-GROUP/hpx/blob/master/.github/CONTRIBUTING.md
[789] http://hpx.stellar-group.org/documents/governance/

4. Build the docs, and proof-read them. Update any documentation that may have changed, and correct any typos. Pay special attention to:

   - `$HPX_SOURCE/README.rst`
     - Update grant information
   - `docs/sphinx/releases/whats_new_$VERSION.rst`
   - `docs/sphinx/about_hpx/people.rst`
     - Update collaborators
     - Update grant information

5. This step does not apply to patch releases. For APEX:

   - Change the release branch to be the most current release tag available in the APEX `git_external` section in the main `CMakeLists.txt`. Please contact the maintainers of the respective packages to generate a new release to synchronize with the *HPX* release (APEX[790]).

6. Make sure `HPX_VERSION_MAJOR/MINOR/SUBMINOR` in `CMakeLists.txt` contain the correct values. Change them if needed.

7. Change version references in CITATION.cff. There are two occurrences. Change year in the copyright file under `/libs/core/version/src/version.cpp`.

8. This step does not apply to patch releases. Remove features which have been deprecated for at least 2 releases. This involves removing build options which enable those features from the main CMakeLists.txt and also deleting all related code and tests from the main source tree.

   The general deprecation policy involves a three-step process we have to go through in order to introduce a breaking change:

   a. First release cycle: add a build option that allows for explicitly disabling any old (now deprecated) code.
   b. Second release cycle: turn this build option OFF by default.
   c. Third release cycle: completely remove the old code.

   The main CMakeLists.txt contains a comment indicating for which version the breaking change was introduced first. In the case of deprecated features which don't have a replacement yet, we keep them around in case (like Vc for example).

9. Update the minimum required versions if necessary (compilers, dependencies, etc.) in `prerequisites.rst`.

10. Verify that the Jenkins setups for the release branch on Rostam and Piz Daint are running and do not display any errors.

11. Repeat the following steps until satisfied with the release.

    1. Change `HPX_VERSION_TAG` in `CMakeLists.txt` to `-rcN`, where `N` is the current iteration of this step. Start with `-rc1`.
    2. Create a pre-release on GitHub using the script `tools/roll_release.sh`. This script automatically tag with the corresponding release number. The script requires that you have the STE||AR Group signing key.
    3. This step is not necessary for patch releases. Notify `hpx-users@stellar-group.org` of the availability of the release candidate. Ask users to test the candidate by checking out the release candidate tag.
    4. Allow at least a week for testing of the release candidate.

[790] http://github.com/UO-OACISS/xpress-apex

   - Use `git merge` when possible, and fall back to `git cherry-pick` when needed. For patch releases `git cherry-pick` is most likely your only choice if there have been significant unrelated changes on master since the previous release.
   - Go back to the first step when enough patches have been added.
   - If there are no more patches, continue to make the final release.

12. Update any occurrences of the latest stable release to refer to the version about to be released. For example, `quickstart.rst` contains instructions to check out the latest stable tag. Make sure that refers to the new version.
13. Add a new entry to the RPM changelog (`cmake/packaging/rpm/Changelog.txt`) with the new version number and a link to the corresponding changelog.
14. Change `HPX_VERSION_TAG` in `CMakeLists.txt` to an empty string.
15. Add the release date to the caption of the current "What's New" section in the docs, and change the value of `HPX_VERSION_DATE` in `CMakeLists.txt`.
16. Create a release on GitHub using the script `tools/roll_release.sh`. This script automatically tag the with the corresponding release number. The script requires that you have the STE||AR Group signing key.
17. Update the websites (hpx.stellar-group.org[791] and *stellar-group.org <https://stellar-group.org>*). You can login on wordpress through *this page <https://hpx.stellar-group.org/wp-login.php>*. You can update the pages with the following:
    - Update links on the downloads page. Link to the release on GitHub.
    - Documentation links on the docs page (link to generated documentation on GitHub Pages). Follow the style of previous releases.
    - A new blog post announcing the release, which links to downloads and the "What's New" section in the documentation (see previous releases for examples).
18. Merge release branch into master.
19. Post-release cleanup. Create a new pull request against master with the following changes:
    1. Modify the release procedure if necessary.
    2. Change `HPX_VERSION_TAG` in `CMakeLists.txt` back to `-trunk`.
    3. Increment `HPX_VERSION_MINOR` in `CMakeLists.txt`.
20. Update Vcpkg (`https://github.com/Microsoft/vcpkg`) to pull from latest release.
    - Update version number in CONTROL
    - Update tag and SHA512 to that of the new release
21. Update spack (`https://github.com/spack/spack`) with the latest HPX package.
    - Update version number in `hpx/package.py` and SHA256 to that of the new release
22. Announce the release on hpx-users@stellar-group.org, stellar@cct.lsu.edu, allcct@cct.lsu.edu, faculty@csc.lsu.edu, faculty@ece.lsu.edu, the *HPX* Slack channel, the IRC channel, our list of external collaborators, isocpp.org, reddit.com, HPC Wire, Inside HPC, Heise Online, and a CCT press release.
23. Beer and pizza.

[791] https://hpx.stellar-group.org

2.9.4 Testing *HPX*

To ensure correctness of *HPX*, we ship a large variety of unit and regression tests. The tests are driven by the CTest[792] tool and are executed automatically on each commit to the *HPX* Github[793] repository. In addition, it is encouraged to run the test suite manually to ensure proper operation on your target system. If a test fails for your platform, we highly recommend submitting an issue on our *HPX* Issues[794] tracker with detailed information about the target system.

Running tests manually

Running the tests manually is as easy as typing `make tests && make test`. This will build all tests and run them once the tests are built successfully. After the tests have been built, you can invoke separate tests with the help of the `ctest` command. You can list all available test targets using `make help | grep tests`. Please see the CTest Documentation[795] for further details.

Running performance tests

We run performance tests on Piz Daint for each pull request using Jenkins. To run those performance tests locally or on Piz Daint, a script is provided under `tools/perftests_ci/local_run.sh` (to be run in the build directory specifying the *HPX* source directory as the argument to the script, default is `$HOME/projects/hpx_perftests_ci`.

Adding new performance tests

To add a new performance test, you need to wrap the portion of code to benchmark with `hpx::util::perftests_report`, passing the test name, the executor name and the function to time (can be a lambda). This facility is used to output the time results in a json format (format needed to compare the results and plot them). To effectively print them at the end of your test, call `hpx::util::perftests_print_times`. To see an example of use, see `future_overhead_report.cpp`. Finally, you can add the test to the CI report editing the `hpx_targets` variable for the executable name and the `hpx_test_options` variable for the corresponding options to use for the run in the performance test script `.jenkins/cscs-perftests/launch_perftests.sh`. And then run the `tools/perftests_ci/local_run.sh` script to get a reference json run (use the name of the test) to be added in the `tools/perftests_ci/perftest/references/daint_default` directory.

Issue tracker

If you stumble over a bug or missing feature in *HPX*, please submit an issue to our *HPX* Issues[796] page. For more information on how to submit support requests or other means of getting in contact with the developers, please see the Support Website[797] page.

[792] https://gitlab.kitware.com/cmake/community/wikis/doc/ctest/Testing-With-CTest
[793] https://github.com/STEllAR-GROUP/hpx/
[794] https://github.com/STEllAR-GROUP/hpx/issues
[795] https://www.cmake.org/cmake/help/latest/manual/ctest.1.html
[796] https://github.com/STEllAR-GROUP/hpx/issues
[797] https://stellar.cct.lsu.edu/support/

Continuous testing

In addition to manual testing, we run automated tests on various platforms. We also run tests on all pull requests using both CircleCI[798] and a combination of CDash[799] and pycicle[800]. You can see the dashboards here: CircleCI HPX dashboard[801] and CDash HPX dashboard[802] .

2.9.5 Using docker for development

Although it can often be useful to set up a local development environment with system-provided or self-built dependencies, Docker[803] provides a convenient alternative to quickly get all the dependencies needed to start development of *HPX*. Our testing setup on CircleCI[804] uses a docker image to run all tests.

To get started you need to install Docker[805] using whatever means is most convenient on your system. Once you have Docker[806] installed, you can pull or directly run the docker image. The image is based on Debian and Clang, and can be found on Docker Hub[807]. To start a container using the *HPX* build environment, run:

```
$ docker run --interactive --tty stellargroup/build_env:latest bash
```

You are now in an environment where all the *HPX* build and runtime dependencies are present. You can install additional packages according to your own needs. Please see the Docker Documentation[808] for more information on using Docker[809].

> **Warning:** All changes made within the container are lost when the container is closed. If you want files to persist (e.g., the *HPX* source tree) after closing the container, you can bind directories from the host system into the container (see Docker Documentation (Bind mounts)[810]).

2.9.6 Documentation

This documentation is built using Sphinx[811], and an automatically generated API reference using Doxygen[812] and Breathe[813].

We always welcome suggestions on how to improve our documentation, as well as pull requests with corrections and additions.

798 https://circleci.com
799 https://www.kitware.com/cdash/project/about.html
800 https://github.com/biddisco/pycicle/
801 https://circleci.com/gh/STEllAR-GROUP/hpx
802 https://cdash.rostam.cct.lsu.edu/index.php?project=HPX
803 https://www.docker.com
804 https://circleci.com
805 https://www.docker.com
806 https://www.docker.com
807 https://hub.docker.com/r/stellargroup/build_env/
808 https://docs.docker.com/
809 https://www.docker.com
810 https://docs.docker.com/storage/bind-mounts/
811 http://www.sphinx-doc.org
812 https://www.doxygen.org
813 https://breathe.readthedocs.io/en/latest

Prerequisites

To build the *HPX* documentation, you need recent versions of the following packages:

- `python3`
- `sphinx 4.5.0` (Python package)
- `sphinx-book-theme` (Python package)
- `breathe 4.33.1` (Python package)
- `doxygen`
- `sphinxcontrib-bibtex`
- `sphinx-copybutton`

If the Python[814] dependencies are not available through your system package manager, you can install them using the Python package manager `pip`:

```
pip install --user "sphinx<5" sphinx-book-theme breathe sphinxcontrib-bibtex sphinx-
↪copybutton
```

You may need to set the following CMake variables to make sure CMake can find the required dependencies.

`Doxygen_ROOT:PATH`
: Specifies where to look for the installation of the Doxygen[815] tool.

`Sphinx_ROOT:PATH`
: Specifies where to look for the installation of the Sphinx[816] tool.

`Breathe_APIDOC_ROOT:PATH`
: Specifies where to look for the installation of the Breathe[817] tool.

Building documentation

Enable building of the documentation by setting `HPX_WITH_DOCUMENTATION=ON` during CMake[818] configuration. To build the documentation, build the `docs` target using your build tool. The default output format is HTML documentation. You can choose alternative output formats (single-page HTML, PDF, and man) with the `HPX_WITH_DOCUMENTATION_OUTPUT_FORMATS` CMake option.

Note: If you add new source files to the Sphinx documentation, you have to run CMake again to have the files included in the build.

[814] https://www.python.org
[815] https://www.doxygen.org
[816] http://www.sphinx-doc.org
[817] https://breathe.readthedocs.io/en/latest
[818] https://www.cmake.org

Style guide

The documentation is written using reStructuredText. These are the conventions used for formatting the documentation:

- Use, at most, 80 characters per line.
- Top-level headings use over- and underlines with =.
- Sub-headings use only underlines with characters in decreasing level of importance: =, - and `.`.
- Use sentence case in headings.
- Refer to common terminology using `` :term:`Component` ``.
- Indent content of directives (`.. directive::`) by three spaces.
- For C++ code samples at the end of paragraphs, use `::` and indent the code sample by 4 spaces.
  - For other languages (or if you don't want a colon at the end of the paragraph), use `.. code-block:: language` and indent by three spaces as with other directives.
- Use `.. list-table::` to wrap tables with a lot of text in cells.

API documentation

The source code is documented using Doxygen. If you add new API documentation either to existing or new source files, make sure that you add the documented source files to the `doxygen_dependencies` variable in `docs/CMakeLists.txt`.

2.9.7 Module structure

This section explains the structure of an *HPX* module.

The tool create_library_skeleton.py[819] can be used to generate a basic skeleton. To create a library skeleton, run the tool in the `libs` subdirectory with the module name as an argument:

```
$ ./create_library_skeleton <lib_name>
```

This creates a skeleton with the necessary files for an *HPX* module. It will not create any actual source files. The structure of this skeleton is as follows:

- <lib_name>/
  - `README.rst`
  - `CMakeLists.txt`
  - `cmake`
  - `docs/`
    - `index.rst`
  - `examples/`
    - `CMakeLists.txt`
  - `include/`
    - `hpx/`

[819] https://github.com/STEllAR-GROUP/hpx/blob/master/libs/create_library_skeleton.py

            · <lib_name>

  – src/

    * CMakeLists.txt

  – tests/

    * CMakeLists.txt

    * unit/

      · CMakeLists.txt

    * regressions/

      · CMakeLists.txt

    * performance/

      · CMakeLists.txt

A `README.rst` should be always included which explains the basic purpose of the library and a link to the generated documentation.

A main `CMakeLists.txt` is created in the root directory of the module. By default it contains a call to `add_hpx_module` which takes care of most of the boilerplate required for a module. You only need to fill in the source and header files in most cases.

`add_hpx_module` requires a module name. Optional flags are:

Optional single-value arguments are:

- `INSTALL_BINARIES`: Install the resulting library.

Optional multi-value arguments-are:

- `SOURCES`: List of source files.
- `HEADERS`: List of header files.
- `COMPAT_HEADERS`: List of compatibility header files.
- `DEPENDENCIES`: Libraries that this module depends on, such as other modules.
- `CMAKE_SUBDIRS`: List of subdirectories to add to the module.

The `include` directory should contain only headers that other libraries need. For each of those headers, an automatic header test to check for self containment will be generated. Private headers should be placed under the `src` directory. This allows for clear separation. The `cmake` subdirectory may include additional CMake[820] scripts needed to generate the respective build configurations.

Compatibility headers (forwarding headers for headers whose location is changed when creating a module, if moving them from the main library) should be placed in an `include_compatibility` directory. This directory is not created by default.

Documentation is placed in the `docs` folder. A empty skeleton for the index is created, which is picked up by the main build system and will be part of the generated documentation. Each header inside the `include` directory will automatically be processed by Doxygen and included into the documentation.

Tests are placed in suitable subdirectories of `tests`.

When in doubt, consult existing modules for examples on how to structure the module.

[820] https://www.cmake.org

Finding circular dependencies

Our CI will perform a check to see if there are circular dependencies between modules. In cases where it's not clear what is causing the circular dependency, running the cpp-dependencies[821] tool manually can be helpful. It can give you detailed information on exactly which files are causing the circular dependency. If you do not have the `cpp-dependencies` tool already installed, one way of obtaining it is by using our docker image. This way you will have exactly the same environment as on the CI. See *Using docker for development* for details on how to use the docker image.

To produce the graph produced by CI run the following command (`HPX_SOURCE` is assumed to hold the path to the *HPX* source directory):

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --graph-cycles circular_dependencies.dot
```

This will produce a `dot` file in the current directory. You can inspect this manually with a text editor. You can also convert this to an image if you have `graphviz` installed:

```
$ dot circular_dependencies.dot -Tsvg -o circular_dependencies.svg
```

This produces an `svg` file in the current directory which shows the circular dependencies. Note that if there are no cycles the image will be empty.

You can use `cpp-dependencies` to print the include paths between two modules.

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --shortest <from> <to>
```

prints all possible paths from the module `<from>` to the module `<to>`. For example, as most modules depend on `config`, the following should give you a long list of paths from `algorithms` to `config`:

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --shortest algorithms config
```

The following should report that it can't find a path between the two modules:

```
$ cpp-dependencies --dir $HPX_SOURCE/libs --shortest config algorithms
```

2.10 Releases

2.10.1 List of releases

HPX V1.10.0 (May 29, 2024)

General changes

- The HPX documentation has seen a major overhaul for this release. We finished documenting the public local HPX API, we have added migration guides from widely used parallelization platforms to HPX (OpenMP, TBB, and MPI).
- We have added facilities enabling optimizations for trivially-relocatable types (see P1144[822] for more details).
- We have added (and use) the `scope_xxx` helper facilities as specified by the C++ library fundamentals TS v3 (see: N4948[823]).

[821] https://github.com/tomtom-international/cpp-dependencies
[822] https://wg21.link/p1144
[823] http://wg21.link/n4948

- We have added configuration options that allow to build HPX without pre-installing any prerequisites. Use `HPX_WITH_FETCH_HWLOC=On` to have Portable Hardware Locality (HWLOC)[824] installed for you. Similarly, setting `HPX_WITH_FETCH_BOOST=On` during configuration time will install the necessary Boost[825] libraries (currently V1.84.0).
- We have performed a lot of code cleanup and refactoring to improve the overall code quality and decrease compile times.
- The collective operations APIs have seen an unification, we have fixed issues and performance problems for the collectives.
- The HPX executors have seen a streamlining and some consistency changes. We have applied many performance improvements to the executor implementations that directly positively impact the performance of our parallel algorithms.
- We have added a new parcelport allowing to use Gasnet as a communication platform.
- We have added optimizations to various parcelports improving overall communication performance. This includes - amongst other things - send immediate optimizations and receiver-side zero-copy optimizations.
- Futures will now execute the associated task eagerly and inline on any wait operation if the task has not started running yet. This feature can be enabled using the `HPX_COROUTINES_WITH_THREAD_SCHEDULE_HINT_RUNS_AS_CHILD=On` configuration setting (which is *Off* by default).
- We have enabled using json files to supply configuration information through the command line. This feature can be enabled with the configuration option `HPX_COMMAND_LINE_HANDLING_WITH_JSON_CONFIGURATION_FILES=On`. This functionality depends on the external JSon library[826], which can be built at configuration time by supplying `HPX_WITH_FETCH_JSON=On` to CMake[827].
- We have applied many fixes to our CUDA, ROCm, and SYCL build environments.

Breaking changes

- The CMake[828] configuration keys `SOMELIB_ROOT` (e.g., `BOOST_ROOT`) have been renamed to `Somelib_ROOT` (e.g., `Boost_ROOT`) to avoid warnings when using newer versions of CMake[829]. Please update your scripts accordingly. For now, the old variable names are re-assigned to the new names and unset in the CMake[830] cache.

Closed issues

- Issue #6466[831] - No access limitations to Wiki
- Issue #6461[832] - handle_received_parcels may never return
- Issue #6459[833] - Building HPX
- Issue #6451[834] - HPX hangs at the very end

824 https://www.open-mpi.org/projects/hwloc/
825 https://www.boost.org/
826 https://github.com/nlohmann/json
827 https://www.cmake.org
828 https://www.cmake.org
829 https://www.cmake.org
830 https://www.cmake.org
831 https://github.com/STEllAR-GROUP/hpx/issues/6466
832 https://github.com/STEllAR-GROUP/hpx/issues/6461
833 https://github.com/STEllAR-GROUP/hpx/issues/6459
834 https://github.com/STEllAR-GROUP/hpx/issues/6451

- Issue #6446[835] - Issue on page /manual/getting_hpx.html
- Issue #6443[836] - PR #6435 (parcel_layer_tweaks) broke Octo-Tiger
- Issue #6440[837] - HPX does not compile with MSVC of Visual Studio 2022 17.9+
- Issue #6437[838] - HPX 1.9.1 does not compile on Fedora with '#pragma message: [Parallel STL message]: "Vectorized algorithm unimplemented, redirected to serial
- Issue #6419[839] - Enhancement of the macro functionalities within hpx
- Issue #6417[840] - The current HPX master branch is still not compatible with Kokkos 4.0.1
- Issue #6414[841] - Current HPX master causes segfaults within Octo-Tiger
- Issue #6412[842] - Clangd (Language Server) throws error for __integer_pack at pack.hpp
- Issue #6407[843] - Cannot build Kokkos 4.0.01 with current HPX master
- Issue #6405[844] - Spack Build Error with ROCm 5.7.0
- Issue #6398[845] - HPX sets affinity wrong with multiple processes per node and LCI parcelport enabled
- Issue #6392[846] - [Feature] Install dependencies using CMake
- Issue #6388[847] - HPX error: "Host not found" when running on Expanse with 128 nodes
- Issue #6366[848] - serialize_buffer allocator support needs adjustments
- Issue #6361[849] - HPX 1.9.1 does not compile on Fedora 40
- Issue #6355[850] - Single page documentation is broken
- Issue #6334[851] - Segmentation fault after adding a padding in one_size_heap_list
- Issue #6329[852] - Log hpx threads on forced shutdown
- Issue #6316[853] - Build breaks on FreeBSD
- Issue #6299[854] - HPX does not use distributed localities on Fugaku
- Issue #6298[855] - Update config for coroutines on ARM
- Issue #6291[856] - Zero-copy receive optimization disabled the invocation of direct actions
- Issue #6261[857] - Add optional reading of json files for command line options

[835] https://github.com/STEllAR-GROUP/hpx/issues/6446
[836] https://github.com/STEllAR-GROUP/hpx/issues/6443
[837] https://github.com/STEllAR-GROUP/hpx/issues/6440
[838] https://github.com/STEllAR-GROUP/hpx/issues/6437
[839] https://github.com/STEllAR-GROUP/hpx/issues/6419
[840] https://github.com/STEllAR-GROUP/hpx/issues/6417
[841] https://github.com/STEllAR-GROUP/hpx/issues/6414
[842] https://github.com/STEllAR-GROUP/hpx/issues/6412
[843] https://github.com/STEllAR-GROUP/hpx/issues/6407
[844] https://github.com/STEllAR-GROUP/hpx/issues/6405
[845] https://github.com/STEllAR-GROUP/hpx/issues/6398
[846] https://github.com/STEllAR-GROUP/hpx/issues/6392
[847] https://github.com/STEllAR-GROUP/hpx/issues/6388
[848] https://github.com/STEllAR-GROUP/hpx/issues/6366
[849] https://github.com/STEllAR-GROUP/hpx/issues/6361
[850] https://github.com/STEllAR-GROUP/hpx/issues/6355
[851] https://github.com/STEllAR-GROUP/hpx/issues/6334
[852] https://github.com/STEllAR-GROUP/hpx/issues/6329
[853] https://github.com/STEllAR-GROUP/hpx/issues/6316
[854] https://github.com/STEllAR-GROUP/hpx/issues/6299
[855] https://github.com/STEllAR-GROUP/hpx/issues/6298
[856] https://github.com/STEllAR-GROUP/hpx/issues/6291
[857] https://github.com/STEllAR-GROUP/hpx/issues/6261

- Issue #6087[858] - Support for vcpkg on Linux is broken
- Issue #5921[859] - hpx::info claims that async_mpi was not built, while cmake assures its existence
- Issue #5893[860] - Tests fail on FreeBSD: Executable copyn_test does not exist
- Issue #5833[861] - barrier lockup
- Issue #5799[862] - Investigate CUDA compilation problems
- Issue #5340[863] - Examples do not run on Mac OSX using the M1 chip

Closed pull requests

- PR #6493[864] - Fix distributed latch documentation
- PR #6492[865] - Fix kokkos hpx nvcc compilation
- PR #6491[866] - More fixes to handling bool arguments for collective operations
- PR #6490[867] - Remove the default max cpu count
- PR #6489[868] - Ensure TCP parcelport is deactivated if not needed
- PR #6488[869] - Fixing handling of bool value type for collective operations
- PR #6485[870] - Destructive interference size
- PR #6484[871] - Improve performance counter error handling
- PR #6482[872] - Generalize the notion of bitwise serialization
- PR #6481[873] - Fixing use of HPX_WITH_CXX_STANDARD
- PR #6480[874] - Remove equal_to from hpx::any
- PR #6479[875] - Remove optimizations for certain built-in compiler intrinsics
- PR #6478[876] - Fixing issues on MacOS
- PR #6477[877] - lci pp: lci's github repo name changed from LC to lci
- PR #6476[878] - Fixing binary filter test target names
- PR #6475[879] - Fix mac os github actions

[858] https://github.com/STEllAR-GROUP/hpx/issues/6087
[859] https://github.com/STEllAR-GROUP/hpx/issues/5921
[860] https://github.com/STEllAR-GROUP/hpx/issues/5893
[861] https://github.com/STEllAR-GROUP/hpx/issues/5833
[862] https://github.com/STEllAR-GROUP/hpx/issues/5799
[863] https://github.com/STEllAR-GROUP/hpx/issues/5340
[864] https://github.com/STEllAR-GROUP/hpx/pull/6493
[865] https://github.com/STEllAR-GROUP/hpx/pull/6492
[866] https://github.com/STEllAR-GROUP/hpx/pull/6491
[867] https://github.com/STEllAR-GROUP/hpx/pull/6490
[868] https://github.com/STEllAR-GROUP/hpx/pull/6489
[869] https://github.com/STEllAR-GROUP/hpx/pull/6488
[870] https://github.com/STEllAR-GROUP/hpx/pull/6485
[871] https://github.com/STEllAR-GROUP/hpx/pull/6484
[872] https://github.com/STEllAR-GROUP/hpx/pull/6482
[873] https://github.com/STEllAR-GROUP/hpx/pull/6481
[874] https://github.com/STEllAR-GROUP/hpx/pull/6480
[875] https://github.com/STEllAR-GROUP/hpx/pull/6479
[876] https://github.com/STEllAR-GROUP/hpx/pull/6478
[877] https://github.com/STEllAR-GROUP/hpx/pull/6477
[878] https://github.com/STEllAR-GROUP/hpx/pull/6476
[879] https://github.com/STEllAR-GROUP/hpx/pull/6475

- PR #6472[880] - Troubleshoot CI hangs
- PR #6469[881] - improve(lci pp): more options to control the LCI parcelport
- PR #6467[882] - Bump jwlawson/actions-setup-cmake from 1.14 to 2.0
- PR #6464[883] - Update docs of "Writing distributed applications" page
- PR #6463[884] - Revert "Always return outermost thread id"
- PR #6458[885] - Reduce test workload to fix CI/CD time-out
- PR #6457[886] - replace boost::array with std::array and update file name
- PR #6456[887] - Move APEX CI to rostam
- PR #6455[888] - Fixing compilation if HPX_HAVE_THREAD_QUEUE_WAITTIME is defined
- PR #6454[889] - Update perftests reference measurements
- PR #6453[890] - Update supported platforms of Manual/Prerequisites page
- PR #6452[891] - Fix nvcc crashes in transform_stream.cu and synchronize.cu
- PR #6450[892] - Fix git tag name in Getting HPX page
- PR #6449[893] - LCI parcelport: add yield to potentially infinite retry loop
- PR #6447[894] - Use compressed ptr in schedulers when 128 atomics are not lockfree
- PR #6445[895] - Fix agas addressing cache
- PR #6444[896] - Update CTestConfig.cmake
- PR #6442[897] - Update CMakeLists.txt
- PR #6441[898] - Minor documentation fixes
- PR #6439[899] - Optimizing use of certain #includes
- PR #6438[900] - Bump jwlawson/actions-setup-cmake from 1.14 to 2.0
- PR #6436[901] - Update docs
- PR #6435[902] - Parcel layer tweaks

[880] https://github.com/STEllAR-GROUP/hpx/pull/6472
[881] https://github.com/STEllAR-GROUP/hpx/pull/6469
[882] https://github.com/STEllAR-GROUP/hpx/pull/6467
[883] https://github.com/STEllAR-GROUP/hpx/pull/6464
[884] https://github.com/STEllAR-GROUP/hpx/pull/6463
[885] https://github.com/STEllAR-GROUP/hpx/pull/6458
[886] https://github.com/STEllAR-GROUP/hpx/pull/6457
[887] https://github.com/STEllAR-GROUP/hpx/pull/6456
[888] https://github.com/STEllAR-GROUP/hpx/pull/6455
[889] https://github.com/STEllAR-GROUP/hpx/pull/6454
[890] https://github.com/STEllAR-GROUP/hpx/pull/6453
[891] https://github.com/STEllAR-GROUP/hpx/pull/6452
[892] https://github.com/STEllAR-GROUP/hpx/pull/6450
[893] https://github.com/STEllAR-GROUP/hpx/pull/6449
[894] https://github.com/STEllAR-GROUP/hpx/pull/6447
[895] https://github.com/STEllAR-GROUP/hpx/pull/6445
[896] https://github.com/STEllAR-GROUP/hpx/pull/6444
[897] https://github.com/STEllAR-GROUP/hpx/pull/6442
[898] https://github.com/STEllAR-GROUP/hpx/pull/6441
[899] https://github.com/STEllAR-GROUP/hpx/pull/6439
[900] https://github.com/STEllAR-GROUP/hpx/pull/6438
[901] https://github.com/STEllAR-GROUP/hpx/pull/6436
[902] https://github.com/STEllAR-GROUP/hpx/pull/6435

- PR #6434[903] - improve termination detection: removing lock from critical path
- PR #6433[904] - Use shared mutex for resolve_locality procedure
- PR #6432[905] - Module cleanup up to level 30
- PR #6429[906] - Making sure HPX_WITH_ASYNC_MPI is reported properly
- PR #6427[907] - Modifying CMakeLists to copy libhwloc-15.dll to the binary folder in Windows, independently
- PR #6425[908] - Fix macOS failing test
- PR #6424[909] - Adding option for downloading Boost using CMake FetchContent
- PR #6423[910] - Move adjacent_difference to numeric header file
- PR #6422[911] - Adding steal-half functionalities to work-requesting scheduler
- PR #6421[912] - Bump actions/checkout from 2 to 4
- PR #6418[913] - Working around nvcc problems to use CTAD
- PR #6416[914] - Change run_as_os_thread deprecation forwarding due to hipcc compilation issue
- PR #6415[915] - Attempting to avoid segfault in OctoTiger during initialization
- PR #6413[916] - Always return outermost thread id
- PR #6411[917] - Minor refactoring and fixes to the LCI parcelport and pingpong_performance2 benchmark
- PR #6410[918] - Adding scope_xxx from library fundamentals TS v3
- PR #6409[919] - Working around CUDA issue
- PR #6408[920] - Tightening up collective operation semantics
- PR #6406[921] - Working around ROCm compiler issue
- PR #6404[922] - Allow to disable use of [[no_unique_address]] attribute
- PR #6403[923] - Fixing copyright year
- PR #6402[924] - fix(lci pp): fix deadlocks with too many failed sends
- PR #6401[925] - fix(lci pp): fix the null_thread_id bug in the LCI parcelport

903 https://github.com/STEllAR-GROUP/hpx/pull/6434
904 https://github.com/STEllAR-GROUP/hpx/pull/6433
905 https://github.com/STEllAR-GROUP/hpx/pull/6432
906 https://github.com/STEllAR-GROUP/hpx/pull/6429
907 https://github.com/STEllAR-GROUP/hpx/pull/6427
908 https://github.com/STEllAR-GROUP/hpx/pull/6425
909 https://github.com/STEllAR-GROUP/hpx/pull/6424
910 https://github.com/STEllAR-GROUP/hpx/pull/6423
911 https://github.com/STEllAR-GROUP/hpx/pull/6422
912 https://github.com/STEllAR-GROUP/hpx/pull/6421
913 https://github.com/STEllAR-GROUP/hpx/pull/6418
914 https://github.com/STEllAR-GROUP/hpx/pull/6416
915 https://github.com/STEllAR-GROUP/hpx/pull/6415
916 https://github.com/STEllAR-GROUP/hpx/pull/6413
917 https://github.com/STEllAR-GROUP/hpx/pull/6411
918 https://github.com/STEllAR-GROUP/hpx/pull/6410
919 https://github.com/STEllAR-GROUP/hpx/pull/6409
920 https://github.com/STEllAR-GROUP/hpx/pull/6408
921 https://github.com/STEllAR-GROUP/hpx/pull/6406
922 https://github.com/STEllAR-GROUP/hpx/pull/6404
923 https://github.com/STEllAR-GROUP/hpx/pull/6403
924 https://github.com/STEllAR-GROUP/hpx/pull/6402
925 https://github.com/STEllAR-GROUP/hpx/pull/6401

- PR #6400[926] - Fix the affinity setting bug when using LCI pp and multiple localities per node
- PR #6397[927] - Change API header titles and info
- PR #6396[928] - Making is_bitwise_serializable SFINAE-friendly
- PR #6395[929] - Adapt amount of collective testing
- PR #6394[930] - Adding option for installing Hwloc using CMake FetchContent
- PR #6393[931] - Optionally disable caching allocator
- PR #6391[932] - Cleaning up collective operations
- PR #6390[933] - Making function local constexpr variables non-static
- PR #6389[934] - Disable resolving hostnames if TCP is disabled
- PR #6387[935] - Need to break out of the loop when searching the suffixes.
- PR #6384[936] - Fixing allocation/deallocation mismatch in serialize_buffer
- PR #6383[937] - Enable fork_join_executor to handle return values from scheduled functions
- PR #6381[938] - Consistently treat conflicting parameters provided by executors and parameter objects
- PR #6380[939] - Fixing setting an annotation for an execution policy
- PR #6378[940] - Allowing to disable signal handlers
- PR #6377[941] - Fix gasnet-related test failures
- PR #6375[942] - Update LSU Jenkins with 2023-10 libraries
- PR #6374[943] - Investigate builder gasnet failure
- PR #6373[944] - Fixing communicator API, adding docs
- PR #6372[945] - Fix resource partitioner tests for small thread count
- PR #6371[946] - Fix jacobi omp examples.
- PR #6370[947] - improve one_size_heap_list: use rwlock to speedup the allocation/free
- PR #6369[948] - working issue with MPI_CC / CC conflict in automake

[926] https://github.com/STEllAR-GROUP/hpx/pull/6400
[927] https://github.com/STEllAR-GROUP/hpx/pull/6397
[928] https://github.com/STEllAR-GROUP/hpx/pull/6396
[929] https://github.com/STEllAR-GROUP/hpx/pull/6395
[930] https://github.com/STEllAR-GROUP/hpx/pull/6394
[931] https://github.com/STEllAR-GROUP/hpx/pull/6393
[932] https://github.com/STEllAR-GROUP/hpx/pull/6391
[933] https://github.com/STEllAR-GROUP/hpx/pull/6390
[934] https://github.com/STEllAR-GROUP/hpx/pull/6389
[935] https://github.com/STEllAR-GROUP/hpx/pull/6387
[936] https://github.com/STEllAR-GROUP/hpx/pull/6384
[937] https://github.com/STEllAR-GROUP/hpx/pull/6383
[938] https://github.com/STEllAR-GROUP/hpx/pull/6381
[939] https://github.com/STEllAR-GROUP/hpx/pull/6380
[940] https://github.com/STEllAR-GROUP/hpx/pull/6378
[941] https://github.com/STEllAR-GROUP/hpx/pull/6377
[942] https://github.com/STEllAR-GROUP/hpx/pull/6375
[943] https://github.com/STEllAR-GROUP/hpx/pull/6374
[944] https://github.com/STEllAR-GROUP/hpx/pull/6373
[945] https://github.com/STEllAR-GROUP/hpx/pull/6372
[946] https://github.com/STEllAR-GROUP/hpx/pull/6371
[947] https://github.com/STEllAR-GROUP/hpx/pull/6370
[948] https://github.com/STEllAR-GROUP/hpx/pull/6369

- PR #6368[949] - Making sure serialize_buffer properly destroys buffer, if needed.
- PR #6367[950] - Fix parallel relocation test
- PR #6364[951] - Relocation variants
- PR #6363[952] - Update the lci parcelport to use LCI v1.7.6
- PR #6362[953] - Fixing compilation problems on 32 Linux systems
- PR #6360[954] - Fix broken links in docs: PDF, Single HTML page, Dependency report
- PR #6359[955] - Fix header file links in Public API page
- PR #6358[956] - Fix CMake find_library for HWLOC
- PR #6357[957] - Replace Custom Benchmarking Code with Nanobench
- PR #6356[958] - Fixed matrix multiplication example output
- PR #6354[959] - Fix broken links for header files in Public API page
- PR #6353[960] - Enable using std::reference_wrapper with executor parameters
- PR #6352[961] - Add Public distributed API documentation
- PR #6350[962] - Make coverage work with Jenkins Github Branch Source plugin
- PR #6349[963] - Moving hpx::threads::run_as_xxx to namespace hpx
- PR #6348[964] - Adding –exclusive to launching tests on rostam
- PR #6346[965] - changed chat link to discord
- PR #6344[966] - uninitialized_relocate w/ type_support primitive
- PR #6343[967] - Bump actions/checkout from 3 to 4
- PR #6342[968] - Fix HPX-APEX cmake integration
- PR #6341[969] - Fix shared_future_continuation_order regression test
- PR #6340[970] - Log alive hpx threads on exit
- PR #6339[971] - Add coverage testing on Jenkins

[949] https://github.com/STEllAR-GROUP/hpx/pull/6368
[950] https://github.com/STEllAR-GROUP/hpx/pull/6367
[951] https://github.com/STEllAR-GROUP/hpx/pull/6364
[952] https://github.com/STEllAR-GROUP/hpx/pull/6363
[953] https://github.com/STEllAR-GROUP/hpx/pull/6362
[954] https://github.com/STEllAR-GROUP/hpx/pull/6360
[955] https://github.com/STEllAR-GROUP/hpx/pull/6359
[956] https://github.com/STEllAR-GROUP/hpx/pull/6358
[957] https://github.com/STEllAR-GROUP/hpx/pull/6357
[958] https://github.com/STEllAR-GROUP/hpx/pull/6356
[959] https://github.com/STEllAR-GROUP/hpx/pull/6354
[960] https://github.com/STEllAR-GROUP/hpx/pull/6353
[961] https://github.com/STEllAR-GROUP/hpx/pull/6352
[962] https://github.com/STEllAR-GROUP/hpx/pull/6350
[963] https://github.com/STEllAR-GROUP/hpx/pull/6349
[964] https://github.com/STEllAR-GROUP/hpx/pull/6348
[965] https://github.com/STEllAR-GROUP/hpx/pull/6346
[966] https://github.com/STEllAR-GROUP/hpx/pull/6344
[967] https://github.com/STEllAR-GROUP/hpx/pull/6343
[968] https://github.com/STEllAR-GROUP/hpx/pull/6342
[969] https://github.com/STEllAR-GROUP/hpx/pull/6341
[970] https://github.com/STEllAR-GROUP/hpx/pull/6340
[971] https://github.com/STEllAR-GROUP/hpx/pull/6339

- PR #6338[972] - Fixing HPX_CURRENT_SOURCE_LOCATION when std::source_location exists
- PR #6337[973] - Remove aurianer, biddisco, and msimberg from codeowners
- PR #6336[974] - More cleaning up for module levels 19-20
- PR #6335[975] - Finalize the MPI docs of the Migration Guide
- PR #6332[976] - More fixes for CMake V3.27
- PR #6330[977] - Adding basic logging to collective operations
- PR #6328[978] - Cleanup previous patch adapting to CMake V3.27
- PR #6327[979] - Modernize modules in level 17 and 18
- PR #6324[980] - P1144 Relocation primitives
- PR #6321[981] - Ensure hpx_main is a proper thread_function
- PR #6320[982] - Fixing cyclic dependencies in naming and agas modules
- PR #6319[983] - Generate git tag if needed but it is not available
- PR #6317[984] - Fixing linker problem on FreeBSD
- PR #6315[985] - acknowledge triv-rel and nothrow-rel types
- PR #6314[986] - Relocation algorithms Clean
- PR #6313[987] - Trivial relocation of c-v-ref-array types
- PR #6312[988] - Fixing warning/error
- PR #6311[989] - Adding executor parallel invoke CPOs
- PR #6310[990] - Define HPX_COMPUTE_CODE in builds with SYCL
- PR #6309[991] - Making sure changed number of cores is propagated to executor
- PR #6308[992] - openshmem-parcelport initial import
- PR #6306[993] - The hpxcxx script was broken such that it could only compile for _release
- PR #6305[994] - Adapting build system for CMake V3.27

[972] https://github.com/STEllAR-GROUP/hpx/pull/6338
[973] https://github.com/STEllAR-GROUP/hpx/pull/6337
[974] https://github.com/STEllAR-GROUP/hpx/pull/6336
[975] https://github.com/STEllAR-GROUP/hpx/pull/6335
[976] https://github.com/STEllAR-GROUP/hpx/pull/6332
[977] https://github.com/STEllAR-GROUP/hpx/pull/6330
[978] https://github.com/STEllAR-GROUP/hpx/pull/6328
[979] https://github.com/STEllAR-GROUP/hpx/pull/6327
[980] https://github.com/STEllAR-GROUP/hpx/pull/6324
[981] https://github.com/STEllAR-GROUP/hpx/pull/6321
[982] https://github.com/STEllAR-GROUP/hpx/pull/6320
[983] https://github.com/STEllAR-GROUP/hpx/pull/6319
[984] https://github.com/STEllAR-GROUP/hpx/pull/6317
[985] https://github.com/STEllAR-GROUP/hpx/pull/6315
[986] https://github.com/STEllAR-GROUP/hpx/pull/6314
[987] https://github.com/STEllAR-GROUP/hpx/pull/6313
[988] https://github.com/STEllAR-GROUP/hpx/pull/6312
[989] https://github.com/STEllAR-GROUP/hpx/pull/6311
[990] https://github.com/STEllAR-GROUP/hpx/pull/6310
[991] https://github.com/STEllAR-GROUP/hpx/pull/6309
[992] https://github.com/STEllAR-GROUP/hpx/pull/6308
[993] https://github.com/STEllAR-GROUP/hpx/pull/6306
[994] https://github.com/STEllAR-GROUP/hpx/pull/6305

- PR #6304[995] - Fixing an integral type mismatch warning
- PR #6303[996] - omp for default vectorization
- PR #6301[997] - Add MPI migration guide
- PR #6294[998] - Add internal reference counting to semaphores
- PR #6286[999] - Simd helpers
- PR #6280[1000] - Add TBB to HPX documentation in Migration Guide
- PR #6276[1001] - Add dependabot.yml
- PR #6275[1002] - Revert "Move dependabot.yml into correct directory"
- PR #6272[1003] - set thread name for linux
- PR #6271[1004] - Uninitialised algorithms, move using std::memcpy
- PR #6270[1005] - Bump jwlawson/actions-setup-cmake from 1.9 to 1.14
- PR #6269[1006] - Bump actions/checkout from 2 to 3
- PR #6268[1007] - Move dependabot.yml into correct directory
- PR #6265[1008] - Create dependabot.yml
- PR #6264[1009] - hpx::is_trivially_relocatable trait implementation
- PR #6263[1010] - Adding support for reading json configuration files for command line options
- PR #6249[1011] - Implement the send immediate optimization for the MPI parcelport.
- PR #6237[1012] - Improve compilation performance
- PR #6234[1013] - Adding release notes page for next release
- PR #6233[1014] - Moving is_relocatable to namespace hpx
- PR #6230[1015] - gasnet based parcelport
- PR #6226[1016] - Re-enable dependency on segmented algorithms on CircleCI
- PR #6220[1017] - Add execution on

[995] https://github.com/STEllAR-GROUP/hpx/pull/6304
[996] https://github.com/STEllAR-GROUP/hpx/pull/6303
[997] https://github.com/STEllAR-GROUP/hpx/pull/6301
[998] https://github.com/STEllAR-GROUP/hpx/pull/6294
[999] https://github.com/STEllAR-GROUP/hpx/pull/6286
[1000] https://github.com/STEllAR-GROUP/hpx/pull/6280
[1001] https://github.com/STEllAR-GROUP/hpx/pull/6276
[1002] https://github.com/STEllAR-GROUP/hpx/pull/6275
[1003] https://github.com/STEllAR-GROUP/hpx/pull/6272
[1004] https://github.com/STEllAR-GROUP/hpx/pull/6271
[1005] https://github.com/STEllAR-GROUP/hpx/pull/6270
[1006] https://github.com/STEllAR-GROUP/hpx/pull/6269
[1007] https://github.com/STEllAR-GROUP/hpx/pull/6268
[1008] https://github.com/STEllAR-GROUP/hpx/pull/6265
[1009] https://github.com/STEllAR-GROUP/hpx/pull/6264
[1010] https://github.com/STEllAR-GROUP/hpx/pull/6263
[1011] https://github.com/STEllAR-GROUP/hpx/pull/6249
[1012] https://github.com/STEllAR-GROUP/hpx/pull/6237
[1013] https://github.com/STEllAR-GROUP/hpx/pull/6234
[1014] https://github.com/STEllAR-GROUP/hpx/pull/6233
[1015] https://github.com/STEllAR-GROUP/hpx/pull/6230
[1016] https://github.com/STEllAR-GROUP/hpx/pull/6226
[1017] https://github.com/STEllAR-GROUP/hpx/pull/6220

- PR #6212[1018] - Initial trait definition for *relocatable*
- PR #6199[1019] - added support for unseq, par_unseq for hpx::make_heap algorithm
- PR #6173[1020] - C++ modules
- PR #6122[1021] - Add Module support
- PR #6099[1022] - Futures attempt to execute threads directly if those have not started executing
- PR #6050[1023] - Investigating partitioned_vector problems
- PR #5988[1024] - Adding CI configuration for DGX-A100 at LSU
- PR #5910[1025] - Improve MPI initialization
- PR #5845[1026] - Adding local work requesting scheduler that is based on message passing internally

HPX V1.9.1 (August 4, 2023)

General changes

This point release fixes a couple of problems reported for the V1.9.0 release. Most importantly, we fixed various occasional hanging during startup and shutdown in distributed scenarios. We also added support for zero-copy serialization on the receiving side to the TCP, MPI, and LCI parcelports. Last but not least, we have added support for Visual Studio 2019 and gcc using MINGW on Windows, and also support for gcc V13 and clang V15.

HPX headers are now made consistently named the same as their standard library counterparts, e.g. *#include <thread>* now corresponds to *#include <hpx/thread.hpp>*. This significantly simplifies porting existing standards conforming codes to HPX.

A lot of work has been done to improve and optimize our network communication layers. Primary focus of this work was on the LCI parcelport, but we have also cleaned up and improved the MPI parcelport.

Additionally, we have continued working on our documentation. The main focus here was on completing the API documentation of the most important API functions. We have started adding migration guides for people interested in moving their codes away from other, commonplace parallelization frameworks like OpenMP.

Breaking changes

None

[1018] https://github.com/STEllAR-GROUP/hpx/pull/6212
[1019] https://github.com/STEllAR-GROUP/hpx/pull/6199
[1020] https://github.com/STEllAR-GROUP/hpx/pull/6173
[1021] https://github.com/STEllAR-GROUP/hpx/pull/6122
[1022] https://github.com/STEllAR-GROUP/hpx/pull/6099
[1023] https://github.com/STEllAR-GROUP/hpx/pull/6050
[1024] https://github.com/STEllAR-GROUP/hpx/pull/5988
[1025] https://github.com/STEllAR-GROUP/hpx/pull/5910
[1026] https://github.com/STEllAR-GROUP/hpx/pull/5845

Closed issues

- Issue #6155[1027] - hpxcxx and hpxrun.py do not work if HPX_WITH_TESTS=OFF
- Issue #6164[1028] - HPX_WITH_DATAPAR_BACKEND=EVE causes compile errors with C++17
- Issue #6175[1029] - Make sure all our parallel algorithms accept the predicates by value
- Issue #6194[1030] - tests.regressions.threads.threads_all_1422 failed at Perlmutter
- Issue #6198[1031] - set_intersection/set_difference fails when run with execution::par
- Issue #6214[1032] - Broken Links to the Documentation page in readme.rst
- Issue #6217[1033] - hpx::make_heap does not terminate when exPolicy is par (or par_unseq) and size of vector is 4
- Issue #6246[1034] - HPX fails to compile under cxx 20 (fresh system)
- Issue #6247[1035] - HPX 1.9.0 does not compile with GCC on Windows
- Issue #6282[1036] - The "attach-debugger" option is broken on the current master branch.

Closed pull requests

- PR #6219[1037] - Cleaning up #includes in hpx/ folder
- PR #6223[1038] - Move documentation from README.rst to index.rst files under libs directory
- PR #6229[1039] - Adding zero-copy support on the receiving end of the TCP and MPI parcel ports
- PR #6231[1040] - Remove deprecated email from release procedure
- PR #6235[1041] - Modernize more modules (levels 12-16)
- PR #6236[1042] - Attempt to resolve occasional shutdown hangs in distributed operation
- PR #6239[1043] - Fix Optimizing HPX applications page of Manual
- PR #6241[1044] - LCI parcelport: Refactor, add more variants, zero copy receives.
- PR #6242[1045] - updated deprecated headers
- PR #6243[1046] - Adding github action builders using VS2019

[1027] https://github.com/STEllAR-GROUP/hpx/issues/6155
[1028] https://github.com/STEllAR-GROUP/hpx/issues/6164
[1029] https://github.com/STEllAR-GROUP/hpx/issues/6175
[1030] https://github.com/STEllAR-GROUP/hpx/issues/6194
[1031] https://github.com/STEllAR-GROUP/hpx/issues/6198
[1032] https://github.com/STEllAR-GROUP/hpx/issues/6214
[1033] https://github.com/STEllAR-GROUP/hpx/issues/6217
[1034] https://github.com/STEllAR-GROUP/hpx/issues/6246
[1035] https://github.com/STEllAR-GROUP/hpx/issues/6247
[1036] https://github.com/STEllAR-GROUP/hpx/issues/6282
[1037] https://github.com/STEllAR-GROUP/hpx/pull/6219
[1038] https://github.com/STEllAR-GROUP/hpx/pull/6223
[1039] https://github.com/STEllAR-GROUP/hpx/pull/6229
[1040] https://github.com/STEllAR-GROUP/hpx/pull/6231
[1041] https://github.com/STEllAR-GROUP/hpx/pull/6235
[1042] https://github.com/STEllAR-GROUP/hpx/pull/6236
[1043] https://github.com/STEllAR-GROUP/hpx/pull/6239
[1044] https://github.com/STEllAR-GROUP/hpx/pull/6241
[1045] https://github.com/STEllAR-GROUP/hpx/pull/6242
[1046] https://github.com/STEllAR-GROUP/hpx/pull/6243

- PR #6248[1047] - Fix CUDA/HIP Jenkins pipelines
- PR #6250[1048] - Resolve gcc problems on Windows
- PR #6251[1049] - Attempting to fix problems in barrier causing hangs
- PR #6253[1050] - Modernize set_thread_name on Windows
- PR #6256[1051] - Fix nvcc/gcc-10 (Octo-Tiger) compilation issue
- PR #6257[1052] - Cmake Tests: Delete operator check for size_t arg
- PR #6258[1053] - Rewriting wait_some to circumvent data races causing hangs
- PR #6260[1054] - Add migration guide to manual
- PR #6262[1055] - Fixing wrong command line options in local command line handling
- PR #6266[1056] - Attempt to resolve occasional hang in run_loop
- PR #6267[1057] - Attempting to fix migration tests
- PR #6278[1058] - Making sure the future's shared state doesn't go out of scope prematurely
- PR #6279[1059] - Re-expose error names
- PR #6281[1060] - Creating directory for file copy
- PR #6283[1061] - Consistently #include unistd.h for _POSIX_VERSION

HPX V1.9.0 (May 2, 2023)

General changes

- Added RISC-V 64bit support. HPX is now compatible with RISC-V architectures which have revolutionized the HPC world.
- LCI parcelport has been optimized to transfer parcels with fewer messages and use the HPX resource partitioner for its progress thread allocation. It should generally provide better performance than before. It also removes its dependency on the MPI library.
- HPX dependency on Boost was further relaxed by replacing headers from Boost.Range, Boost.Tokenizer and Boost.Lockfree.
- Improvements took place on our parallel algorithms implementation.
- Our Senders/Receivers (P2300) integration was extended:
  - Coroutines were integrated with senders/receivers.

[1047] https://github.com/STEllAR-GROUP/hpx/pull/6248
[1048] https://github.com/STEllAR-GROUP/hpx/pull/6250
[1049] https://github.com/STEllAR-GROUP/hpx/pull/6251
[1050] https://github.com/STEllAR-GROUP/hpx/pull/6253
[1051] https://github.com/STEllAR-GROUP/hpx/pull/6256
[1052] https://github.com/STEllAR-GROUP/hpx/pull/6257
[1053] https://github.com/STEllAR-GROUP/hpx/pull/6258
[1054] https://github.com/STEllAR-GROUP/hpx/pull/6260
[1055] https://github.com/STEllAR-GROUP/hpx/pull/6262
[1056] https://github.com/STEllAR-GROUP/hpx/pull/6266
[1057] https://github.com/STEllAR-GROUP/hpx/pull/6267
[1058] https://github.com/STEllAR-GROUP/hpx/pull/6278
[1059] https://github.com/STEllAR-GROUP/hpx/pull/6279
[1060] https://github.com/STEllAR-GROUP/hpx/pull/6281
[1061] https://github.com/STEllAR-GROUP/hpx/pull/6283

get_completion_signatures now works with awaitable senders. - with_awaitable_senders allows the passed senders to retrieve the value i.e. senders are transparently awaitable from within a coroutine. - when_all_vector was added.

- sync_wait and sync_wait_with_variant sender consumers were added. The user can now initiate the execution of their asynchronous pipeline by blocking the current thread that executes the main() function until the result is retrieved.
- The combinators for futures (a.k.a. async_combinators) when_*, wait_*, wait_*_nothrow were turned into CPOs allowing for end-user customization. For more information on the async_combinators refer to the documentation, https://hpx-docs.stellar-group.org/latest/html/libs/core/async_combinators/docs/index.html?highlight=combinators.
- The new datapar backend SVE allows simd and par_simd execution policies to exploit dataparalleism in the processors that have SVE vector registers like A64FX and Neoverse V1.
- The documentation for parallel algorithms, container algorithms was further improved. The Public API page was vastly enriched.
- Copy button shortkey was added at the top-right of code-blocks.
- Pragma directive that reports warnings as errors on MSVC was fixed.
- Command line argument --hpx:loopback_network was added to facilitie debugging with networks.
- We added an HPX-SYCL integration, allowing users to obtain HPX futures for SYCL events. This effectively enables the integration of arbitrary asynchronous SYCL operations into the HPX task graph. Bolted on top of this integration, we further added an HPX-SYCL executor for ease of use.

Breaking changes

- Stopped supporting Clang V8, the minimal version supported is now Clang V10.
- Stopped supporting gcc V8, the minimal version supported is now gcc V9.
- Stopped supporting Visual Studio 2015, the minimal version supported is now Visual Studio 2019.
- tag_policy_tag et.al. were re-added after HPX V1.8.1 depracation.
- get_chunk_size and processing_units_count API is now expecting the time for one iteration as an argument.
- The list of all the namespace changes can be found here: *HPX V1.9.0 Namespace changes*.

Closed issues

- Issue #6203[1062] - Compilation error with *-mcpu=a64fx* on Ookami
- Issue #6196[1063] - Incorrect log destination
- Issue #6191[1064] - installing HPX
- Issue #6184[1065] - Wrong processing_units_count of restricted_thread_pool_executor
- Issue #6171[1066] - Release Tag Name Request

[1062] https://github.com/STEllAR-GROUP/hpx/issues/6203
[1063] https://github.com/STEllAR-GROUP/hpx/issues/6196
[1064] https://github.com/STEllAR-GROUP/hpx/issues/6191
[1065] https://github.com/STEllAR-GROUP/hpx/issues/6184
[1066] https://github.com/STEllAR-GROUP/hpx/issues/6171

- Issue #6162[1067] - Current master does not compile on ROSTAM
- Issue #6156[1068] - hpxcxx does not work if HPX_WITH_PKGCONFIG=OFF
- Issue #6108[1069] - cxx17_aligned_new.cpp on msvc fails due to wrong pragma directive
- Issue #6045[1070] - Can't call nullary callables wrapped with *hpx::unwrapping*
- Issue #6013[1071] - Unable to build subprojects hpx_collectives/hpx_compute with MSVC
- Issue #6008[1072] - Missing *constexpr* default constructor for *hpx::mutex*
- Issue #5999[1073] - Add HPX Conda package to conda-forge
- Issue #5998[1074] - Serializing multiple arguments when applying distributed action results in segfault
- Issue #5958[1075] - HPX 1.8.0 and Blaze issues
- Issue #5908[1076] - Windows: duplicated symbols in static builds
- Issue #5802[1077] - Lost status is_ready from future
- Issue #5767[1078] - Performance drop on Piz Daint
- Issue #5752[1079] - Implement stride_view from P1899 (experimental)
- Issue #5744[1080] - HPX_WITH_FETCH_ASIO not working on Ookami
- Issue #5561[1081] - Possible race condition in helper thread / hpx::cout

Closed pull requests

- PR #6228[1082] - Fixing algorithms for zero length sequences when run with s/r scheduler
- PR #6227[1083] - Reliably disable background work when no networking is enabled
- PR #6225[1084] - Make heap fails in par for small sized heaps #6217
- PR #6222[1085] - Add documentation for *hpx::post*
- PR #6221[1086] - Fix segmented algorithms tests
- PR #6218[1087] - Creating INSTALL component 'runtime' to enable installing binaries only
- PR #6216[1088] - added tests for set_difference, updated set_operation.hpp to fix #6198

1067 https://github.com/STEllAR-GROUP/hpx/issues/6162
1068 https://github.com/STEllAR-GROUP/hpx/issues/6156
1069 https://github.com/STEllAR-GROUP/hpx/issues/6108
1070 https://github.com/STEllAR-GROUP/hpx/issues/6045
1071 https://github.com/STEllAR-GROUP/hpx/issues/6013
1072 https://github.com/STEllAR-GROUP/hpx/issues/6008
1073 https://github.com/STEllAR-GROUP/hpx/issues/5999
1074 https://github.com/STEllAR-GROUP/hpx/issues/5998
1075 https://github.com/STEllAR-GROUP/hpx/issues/5958
1076 https://github.com/STEllAR-GROUP/hpx/issues/5908
1077 https://github.com/STEllAR-GROUP/hpx/issues/5802
1078 https://github.com/STEllAR-GROUP/hpx/issues/5767
1079 https://github.com/STEllAR-GROUP/hpx/issues/5752
1080 https://github.com/STEllAR-GROUP/hpx/issues/5744
1081 https://github.com/STEllAR-GROUP/hpx/issues/5561
1082 https://github.com/STEllAR-GROUP/hpx/pull/6228
1083 https://github.com/STEllAR-GROUP/hpx/pull/6227
1084 https://github.com/STEllAR-GROUP/hpx/pull/6225
1085 https://github.com/STEllAR-GROUP/hpx/pull/6222
1086 https://github.com/STEllAR-GROUP/hpx/pull/6221
1087 https://github.com/STEllAR-GROUP/hpx/pull/6218
1088 https://github.com/STEllAR-GROUP/hpx/pull/6216

- PR #6213[1089] - Modernize and streamline MPI parcelport
- PR #6211[1090] - Modernize modules of level 11, 12, and 13
- PR #6210[1091] - Fixing MPI parcelport initialization if MPI is initialized outside of HPX
- PR #6209[1092] - Prevent thread stealing during scheduler shutdown
- PR #6208[1093] - Fix the compilation warning in the MPI parcelport with gcc 11.2
- PR #6207[1094] - Automatically enable Boost.Context when compiling for arm64.
- PR #6206[1095] - Update CMakeLists.txt
- PR #6205[1096] - Do not generate hpxcxx if support for pkgconfig was disabled
- PR #6204[1097] - Use **LRT_** instead of **LAPP_** logging in barrier implementation
- PR #6202[1098] - Fixing Fedora build errors on Power systems
- PR #6201[1099] - Update the LCI parcelport documents
- PR #6200[1100] - Par link jobs
- PR #6197[1101] - LCI parcelport: add doc, upgrade to v1.7.4, refactor cmake autofetch.
- PR #6195[1102] - Change the default tag of autofetch LCI to v1.7.3.
- PR #6192[1103] - Fix page *Writing single-node applications*
- PR #6189[1104] - Making sure restricted_thread_pool_executor properly reports used number of cores
- PR #6187[1105] - Enable using for_loop with range generators
- PR #6186[1106] - thread_support/CMakeLists: Fix build issue
- PR #6185[1107] - Fix EVE datapar with cxx_standard less than 20
- PR #6183[1108] - Update CI integration for EVE
- PR #6182[1109] - Fixing performance regressions
- PR #6181[1110] - LCI parcelport: backlog queue, aggregation, separate devices, and more
- PR #6180[1111] - Fixing use of for_loop with rebound execution policy (using *.with()*)

1089 https://github.com/STEllAR-GROUP/hpx/pull/6213
1090 https://github.com/STEllAR-GROUP/hpx/pull/6211
1091 https://github.com/STEllAR-GROUP/hpx/pull/6210
1092 https://github.com/STEllAR-GROUP/hpx/pull/6209
1093 https://github.com/STEllAR-GROUP/hpx/pull/6208
1094 https://github.com/STEllAR-GROUP/hpx/pull/6207
1095 https://github.com/STEllAR-GROUP/hpx/pull/6206
1096 https://github.com/STEllAR-GROUP/hpx/pull/6205
1097 https://github.com/STEllAR-GROUP/hpx/pull/6204
1098 https://github.com/STEllAR-GROUP/hpx/pull/6202
1099 https://github.com/STEllAR-GROUP/hpx/pull/6201
1100 https://github.com/STEllAR-GROUP/hpx/pull/6200
1101 https://github.com/STEllAR-GROUP/hpx/pull/6197
1102 https://github.com/STEllAR-GROUP/hpx/pull/6195
1103 https://github.com/STEllAR-GROUP/hpx/pull/6192
1104 https://github.com/STEllAR-GROUP/hpx/pull/6189
1105 https://github.com/STEllAR-GROUP/hpx/pull/6187
1106 https://github.com/STEllAR-GROUP/hpx/pull/6186
1107 https://github.com/STEllAR-GROUP/hpx/pull/6185
1108 https://github.com/STEllAR-GROUP/hpx/pull/6183
1109 https://github.com/STEllAR-GROUP/hpx/pull/6182
1110 https://github.com/STEllAR-GROUP/hpx/pull/6181
1111 https://github.com/STEllAR-GROUP/hpx/pull/6180

- PR #6179[1112] - Taking predicates for algorithms by value
- PR #6178[1113] - Changes needed to make chapel_hpx examples work
- PR #6176[1114] - Fixing warnings that were generated by PVS Studio
- PR #6174[1115] - Replace boost::integer::gcd with std::gcd
- PR #6172[1116] - [Docs] Fix example of how to run single/specific test(s)
- PR #6170[1117] - Adding missing fallback for processing_units_count customization point
- PR #6169[1118] - LCI parcelport: bypass the parcel queue and connection cache.
- PR #6167[1119] - Add create_local_communicator API function
- PR #6166[1120] - Add missing header for std::intmax_t
- PR #6165[1121] - Attempt to work around MSVC problem
- PR #6161[1122] - Update EVE integration
- PR #6160[1123] - More cleanup for module levels 0 to 10
- PR #6159[1124] - Fix minor spelling mistake in generate_issue_pr_list.sh
- PR #6158[1125] - Update documentation in *writing single-node applications* page
- PR #6157[1126] - Improve index_queue_spawning
- PR #6154[1127] - Avoid performing late command line handling twice in distributed runtime
- PR #6152[1128] - The -rd and -mr options didn't work, and they should have been –rd and –mr
- PR #6151[1129] - Refactoring the Manual page in documentation
- PR #6148[1130] - Investigate the failure of the LCI parcelport.
- PR #6147[1131] - Make posix co-routine stacks non-executable
- PR #6146[1132] - Avoid ambiguities wrt tag_invoke
- PR #6144[1133] - General improvements to scheduling and related fixes
- PR #6143[1134] - Add list of new namespaces for new release

[1112] https://github.com/STEllAR-GROUP/hpx/pull/6179
[1113] https://github.com/STEllAR-GROUP/hpx/pull/6178
[1114] https://github.com/STEllAR-GROUP/hpx/pull/6176
[1115] https://github.com/STEllAR-GROUP/hpx/pull/6174
[1116] https://github.com/STEllAR-GROUP/hpx/pull/6172
[1117] https://github.com/STEllAR-GROUP/hpx/pull/6170
[1118] https://github.com/STEllAR-GROUP/hpx/pull/6169
[1119] https://github.com/STEllAR-GROUP/hpx/pull/6167
[1120] https://github.com/STEllAR-GROUP/hpx/pull/6166
[1121] https://github.com/STEllAR-GROUP/hpx/pull/6165
[1122] https://github.com/STEllAR-GROUP/hpx/pull/6161
[1123] https://github.com/STEllAR-GROUP/hpx/pull/6160
[1124] https://github.com/STEllAR-GROUP/hpx/pull/6159
[1125] https://github.com/STEllAR-GROUP/hpx/pull/6158
[1126] https://github.com/STEllAR-GROUP/hpx/pull/6157
[1127] https://github.com/STEllAR-GROUP/hpx/pull/6154
[1128] https://github.com/STEllAR-GROUP/hpx/pull/6152
[1129] https://github.com/STEllAR-GROUP/hpx/pull/6151
[1130] https://github.com/STEllAR-GROUP/hpx/pull/6148
[1131] https://github.com/STEllAR-GROUP/hpx/pull/6147
[1132] https://github.com/STEllAR-GROUP/hpx/pull/6146
[1133] https://github.com/STEllAR-GROUP/hpx/pull/6144
[1134] https://github.com/STEllAR-GROUP/hpx/pull/6143

- PR #6140[1135] - Fixing background scheduler to properly exit in the end
- PR #6139[1136] - [P2300] execution: Cleanup coroutines integration and improve ADL isolation
- PR #6137[1137] - Adding example of a simple master/slave distributed application
- PR #6136[1138] - Deprecate *execution::experimental::task_group* in favor of *experimental::task_group*
- PR #6135[1139] - Fixing warnings reported by MSVC analysis
- PR #6134[1140] - Adding notification function for parcelports to be called after early parcel handling
- PR #6132[1141] - Fixing to_non_par() for parallel simd policies
- PR #6131[1142] - modernize modules from level 25
- PR #6130[1143] - Remove the mutex lock in the critical path of get_partitioner.
- PR #6129[1144] - Modernize module from levels 22, 23
- PR #6127[1145] - Working around gccV9 problem that prevent us from storing enum classes in bit fields
- PR #6126[1146] - Deprecate hpx::parallel::task_block in favor of hpx::experimental::ta…
- PR #6125[1147] - Making sure sync_wait compiles when used with an lvalue sender involving bulk
- PR #6124[1148] - Fixing use of any_sender in combination with when_all
- PR #6123[1149] - Fixed issues found by PVS-Studio
- PR #6121[1150] - Modernize modules of level 21, 22
- PR #6120[1151] - Use index_queue for parallel executors bulk_async_execute
- PR #6119[1152] - Update CMakeLists.txt
- PR #6118[1153] - Modernize modules from level 17, 18, 19, and 20
- PR #6117[1154] - Initialize **buffer_allocate_time_** to 0
- PR #6116[1155] - Add new command line argument –hpx:loopback_network
- PR #6115[1156] - Modernize modules of levels 14, 15, and 16
- PR #6114[1157] - Enhance the formatting of the documentation

[1135] https://github.com/STEllAR-GROUP/hpx/pull/6140
[1136] https://github.com/STEllAR-GROUP/hpx/pull/6139
[1137] https://github.com/STEllAR-GROUP/hpx/pull/6137
[1138] https://github.com/STEllAR-GROUP/hpx/pull/6136
[1139] https://github.com/STEllAR-GROUP/hpx/pull/6135
[1140] https://github.com/STEllAR-GROUP/hpx/pull/6134
[1141] https://github.com/STEllAR-GROUP/hpx/pull/6132
[1142] https://github.com/STEllAR-GROUP/hpx/pull/6131
[1143] https://github.com/STEllAR-GROUP/hpx/pull/6130
[1144] https://github.com/STEllAR-GROUP/hpx/pull/6129
[1145] https://github.com/STEllAR-GROUP/hpx/pull/6127
[1146] https://github.com/STEllAR-GROUP/hpx/pull/6126
[1147] https://github.com/STEllAR-GROUP/hpx/pull/6125
[1148] https://github.com/STEllAR-GROUP/hpx/pull/6124
[1149] https://github.com/STEllAR-GROUP/hpx/pull/6123
[1150] https://github.com/STEllAR-GROUP/hpx/pull/6121
[1151] https://github.com/STEllAR-GROUP/hpx/pull/6120
[1152] https://github.com/STEllAR-GROUP/hpx/pull/6119
[1153] https://github.com/STEllAR-GROUP/hpx/pull/6118
[1154] https://github.com/STEllAR-GROUP/hpx/pull/6117
[1155] https://github.com/STEllAR-GROUP/hpx/pull/6116
[1156] https://github.com/STEllAR-GROUP/hpx/pull/6115
[1157] https://github.com/STEllAR-GROUP/hpx/pull/6114

- PR #6113[1158] - Modernize modules in module level 11, 12, and 13
- PR #6112[1159] - Modernize modules from levels 9 and 10
- PR #6111[1160] - Modernize all modules from module level 8
- PR #6110[1161] - Use pragma error directive to report warnings as errors on msvc
- PR #6109[1162] - Modernize serialization module
- PR #6107[1163] - Modernize error module
- PR #6106[1164] - Modernizing modules of levels 0 to 5
- PR #6105[1165] - Optimizations on LCI parcelport: merge small messages; remove sender mutex lock.
- PR #6104[1166] - Adding parameters API: measure_iteration
- PR #6103[1167] - Document *task_group* and include in Public API
- PR #6102[1168] - Prevent warnings generated by clang-cl
- PR #6101[1169] - Using more fold expressions
- PR #6100[1170] - Deprecate *hpx::parallel::reduce_by_key* in favor of *hpx::experimental::reduce_by_key*
- PR #6098[1171] - Forking Boost.Lockfree
- PR #6096[1172] - Forking Boost.Tokenizer
- PR #6095[1173] - Replacing facilities from Boost.Range
- PR #6094[1174] - Removing object_semaphore
- PR #6093[1175] - Replace boost::string_ref with std::string_view
- PR #6092[1176] - Use C++17 static_assert where possible
- PR #6091[1177] - Replace artificial sequencing with fold expressions
- PR #6090[1178] - Fixing use of get_chunk_size customization point
- PR #6088[1179] - Add/fix Public API documentation
- PR #6086[1180] - Deprecate *hpx::util::unlock_guard* in favor of *hpx::unlock_guard*

[1158] https://github.com/STEllAR-GROUP/hpx/pull/6113
[1159] https://github.com/STEllAR-GROUP/hpx/pull/6112
[1160] https://github.com/STEllAR-GROUP/hpx/pull/6111
[1161] https://github.com/STEllAR-GROUP/hpx/pull/6110
[1162] https://github.com/STEllAR-GROUP/hpx/pull/6109
[1163] https://github.com/STEllAR-GROUP/hpx/pull/6107
[1164] https://github.com/STEllAR-GROUP/hpx/pull/6106
[1165] https://github.com/STEllAR-GROUP/hpx/pull/6105
[1166] https://github.com/STEllAR-GROUP/hpx/pull/6104
[1167] https://github.com/STEllAR-GROUP/hpx/pull/6103
[1168] https://github.com/STEllAR-GROUP/hpx/pull/6102
[1169] https://github.com/STEllAR-GROUP/hpx/pull/6101
[1170] https://github.com/STEllAR-GROUP/hpx/pull/6100
[1171] https://github.com/STEllAR-GROUP/hpx/pull/6098
[1172] https://github.com/STEllAR-GROUP/hpx/pull/6096
[1173] https://github.com/STEllAR-GROUP/hpx/pull/6095
[1174] https://github.com/STEllAR-GROUP/hpx/pull/6094
[1175] https://github.com/STEllAR-GROUP/hpx/pull/6093
[1176] https://github.com/STEllAR-GROUP/hpx/pull/6092
[1177] https://github.com/STEllAR-GROUP/hpx/pull/6091
[1178] https://github.com/STEllAR-GROUP/hpx/pull/6090
[1179] https://github.com/STEllAR-GROUP/hpx/pull/6088
[1180] https://github.com/STEllAR-GROUP/hpx/pull/6086

- PR #6085[1181] - Add experimental sycl integration/executor
- PR #6084[1182] - Renaming hpx::apply and friends to hpx::post
- PR #6083[1183] - Using if constexpr instead of tag-dispatching, where possible
- PR #6082[1184] - Replace util::always_void_t with std::void_t
- PR #6081[1185] - Update github actions to avoid warnings
- PR #6080[1186] - Disable some tests that fail on LCI
- PR #6079[1187] - Adding more natvis files, correct existing
- PR #6078[1188] - Changing target name of memory_counters component
- PR #6077[1189] - Making default constructor of hpx::mutex constexpr
- PR #6076[1190] - Cleaning up functionality that was deprecated in V1.7
- PR #6075[1191] - Remove conditional code for gcc V7 and below
- PR #6074[1192] - Fixing compilation issues on gcc V8
- PR #6073[1193] - Fixing PAPI counter component compilation
- PR #6072[1194] - Adding ex::when_all_vector
- PR #6071[1195] - Making get_forward_progress_guarantee_t specializations constexpr
- PR #6070[1196] - Implement P2690 for our algorithms
- PR #6069[1197] - Do not check for cancellation during each iteration but only once per partition
- PR #6068[1198] - Prevent using task and non_task as a CPO
- PR #6067[1199] - Deprecated hpx::util::mem_fn in favor of hpx::mem_fn
- PR #6066[1200] - Create codeql.yml
- PR #6064[1201] - Adapting adjacent_difference for S/R execution
- PR #6063[1202] - Modernize iterator_support module
- PR #6062[1203] - Make sure wrapping executor does not go out of scope prematurely

[1181] https://github.com/STEllAR-GROUP/hpx/pull/6085
[1182] https://github.com/STEllAR-GROUP/hpx/pull/6084
[1183] https://github.com/STEllAR-GROUP/hpx/pull/6083
[1184] https://github.com/STEllAR-GROUP/hpx/pull/6082
[1185] https://github.com/STEllAR-GROUP/hpx/pull/6081
[1186] https://github.com/STEllAR-GROUP/hpx/pull/6080
[1187] https://github.com/STEllAR-GROUP/hpx/pull/6079
[1188] https://github.com/STEllAR-GROUP/hpx/pull/6078
[1189] https://github.com/STEllAR-GROUP/hpx/pull/6077
[1190] https://github.com/STEllAR-GROUP/hpx/pull/6076
[1191] https://github.com/STEllAR-GROUP/hpx/pull/6075
[1192] https://github.com/STEllAR-GROUP/hpx/pull/6074
[1193] https://github.com/STEllAR-GROUP/hpx/pull/6073
[1194] https://github.com/STEllAR-GROUP/hpx/pull/6072
[1195] https://github.com/STEllAR-GROUP/hpx/pull/6071
[1196] https://github.com/STEllAR-GROUP/hpx/pull/6070
[1197] https://github.com/STEllAR-GROUP/hpx/pull/6069
[1198] https://github.com/STEllAR-GROUP/hpx/pull/6068
[1199] https://github.com/STEllAR-GROUP/hpx/pull/6067
[1200] https://github.com/STEllAR-GROUP/hpx/pull/6066
[1201] https://github.com/STEllAR-GROUP/hpx/pull/6064
[1202] https://github.com/STEllAR-GROUP/hpx/pull/6063
[1203] https://github.com/STEllAR-GROUP/hpx/pull/6062

- PR #6061[1204] - Minor fix in small_vector (from upstream)
- PR #6060[1205] - Allow to disable registering signal handlers
- PR #6059[1206] - [P2300] Fix: declval cannot be ODR used
- PR #6058[1207] - Avoid ambiguity for hpx::get used with std::variant
- PR #6057[1208] - Create a dedicated thread pool to run LCI_progress.
- PR #6056[1209] - Fix coroutine test for clang
- PR #6055[1210] - Patches needed to be able to build HPX 1.8.1 on various platforms
- PR #6054[1211] - Use MSVC specific attribute [[msvc::no_unique_address]]
- PR #6052[1212] - Deprecated hpx::util::invoke_fused in favor of hpx::invoke_fused
- PR #6051[1213] - Add non-contiguous index queue and use it in thread_pool_bulk_scheduler
- PR #6049[1214] - Crosscompile arm sve
- PR #6048[1215] - Deprecated hpx::util::invoke in favor of hpx::invoke
- PR #6047[1216] - Separating binary_semaphore into its own file
- PR #6046[1217] - Support using unwrapping with nullary function objects
- PR #6044[1218] - Generalize the use of then() and dataflow
- PR #6043[1219] - Clean up scan_partitioner
- PR #6042[1220] - Modernize dataflow API
- PR #6041[1221] - docs: document semaphores
- PR #6040[1222] - Add/Fix documentation of Public API page
- PR #6039[1223] - remove MPI dependency when only using LCI parcelport
- PR #6038[1224] - Clean up command line handling
- PR #6037[1225] - Avoid performing parcel related background work if networking is disabled
- PR #6036[1226] - Support new datapar backend : SVE

[1204] https://github.com/STEllAR-GROUP/hpx/pull/6061
[1205] https://github.com/STEllAR-GROUP/hpx/pull/6060
[1206] https://github.com/STEllAR-GROUP/hpx/pull/6059
[1207] https://github.com/STEllAR-GROUP/hpx/pull/6058
[1208] https://github.com/STEllAR-GROUP/hpx/pull/6057
[1209] https://github.com/STEllAR-GROUP/hpx/pull/6056
[1210] https://github.com/STEllAR-GROUP/hpx/pull/6055
[1211] https://github.com/STEllAR-GROUP/hpx/pull/6054
[1212] https://github.com/STEllAR-GROUP/hpx/pull/6052
[1213] https://github.com/STEllAR-GROUP/hpx/pull/6051
[1214] https://github.com/STEllAR-GROUP/hpx/pull/6049
[1215] https://github.com/STEllAR-GROUP/hpx/pull/6048
[1216] https://github.com/STEllAR-GROUP/hpx/pull/6047
[1217] https://github.com/STEllAR-GROUP/hpx/pull/6046
[1218] https://github.com/STEllAR-GROUP/hpx/pull/6044
[1219] https://github.com/STEllAR-GROUP/hpx/pull/6043
[1220] https://github.com/STEllAR-GROUP/hpx/pull/6042
[1221] https://github.com/STEllAR-GROUP/hpx/pull/6041
[1222] https://github.com/STEllAR-GROUP/hpx/pull/6040
[1223] https://github.com/STEllAR-GROUP/hpx/pull/6039
[1224] https://github.com/STEllAR-GROUP/hpx/pull/6038
[1225] https://github.com/STEllAR-GROUP/hpx/pull/6037
[1226] https://github.com/STEllAR-GROUP/hpx/pull/6036

- PR #6035[1227] - Simplify datapar replace copy if
- PR #6034[1228] - Add/Fix documentation of Public API
- PR #6033[1229] - Support for data-parallelism for replace, replace_if, replace_copy, replace_copy_if algorithms
- PR #6032[1230] - Add documentation in public API
- PR #6031[1231] - Expose available cache sizes from topology object
- PR #6030[1232] - Adding parcelport initialization hook for resource partitioner operation
- PR #6029[1233] - Simplify startup code
- PR #6027[1234] - Add/Fix documentation in Public API page
- PR #6026[1235] - add option hpx:force_ipv4 to force resolving hostnames to ipv4 adresses
- PR #6025[1236] - build(docs): remove leftover sections
- PR #6023[1237] - Minor fixes on "How to build on Windows"
- PR #6022[1238] - build(doxy): don't extract private members
- PR #6021[1239] - Adding pu_mask to thread_pool_bulk_scheduler
- PR #6020[1240] - docs: add cppref NamedRequirements support
- PR #6018[1241] - Unseq adaptation for for_each, transform, reduce, transform_reduce, etc.
- PR #6017[1242] - loop and transform_loop unseq adaptation
- PR #6016[1243] - Config and structural updates to support unseq implementation
- PR #6015[1244] - Integrating sync_wait & sync_wait_with_variant
- PR #6012[1245] - docs: add missing links to public api
- PR #6009[1246] - Fixing sender&receiver integration with for_each and for_loop
- PR #6007[1247] - docs: add docs for mutex.hpp
- PR #6006[1248] - Relax future::is_ready where possible
- PR #6005[1249] - reshuffle header tests to different instances

[1227] https://github.com/STEllAR-GROUP/hpx/pull/6035
[1228] https://github.com/STEllAR-GROUP/hpx/pull/6034
[1229] https://github.com/STEllAR-GROUP/hpx/pull/6033
[1230] https://github.com/STEllAR-GROUP/hpx/pull/6032
[1231] https://github.com/STEllAR-GROUP/hpx/pull/6031
[1232] https://github.com/STEllAR-GROUP/hpx/pull/6030
[1233] https://github.com/STEllAR-GROUP/hpx/pull/6029
[1234] https://github.com/STEllAR-GROUP/hpx/pull/6027
[1235] https://github.com/STEllAR-GROUP/hpx/pull/6026
[1236] https://github.com/STEllAR-GROUP/hpx/pull/6025
[1237] https://github.com/STEllAR-GROUP/hpx/pull/6023
[1238] https://github.com/STEllAR-GROUP/hpx/pull/6022
[1239] https://github.com/STEllAR-GROUP/hpx/pull/6021
[1240] https://github.com/STEllAR-GROUP/hpx/pull/6020
[1241] https://github.com/STEllAR-GROUP/hpx/pull/6018
[1242] https://github.com/STEllAR-GROUP/hpx/pull/6017
[1243] https://github.com/STEllAR-GROUP/hpx/pull/6016
[1244] https://github.com/STEllAR-GROUP/hpx/pull/6015
[1245] https://github.com/STEllAR-GROUP/hpx/pull/6012
[1246] https://github.com/STEllAR-GROUP/hpx/pull/6009
[1247] https://github.com/STEllAR-GROUP/hpx/pull/6007
[1248] https://github.com/STEllAR-GROUP/hpx/pull/6006
[1249] https://github.com/STEllAR-GROUP/hpx/pull/6005

- PR #6004[1250] - Add documentation Public API
- PR #6003[1251] - Always exporting get_component_name implementations
- PR #6002[1252] - Making sure that default constructble arguments are properly constructed during deserialization
- PR #5996[1253] - Add back explicit template parameters to lock_guards for nvcc
- PR #5994[1254] - Fix CTRL+C on windows
- PR #5993[1255] - Using EVE requires C++20
- PR #5992[1256] - This properly terminates an application on Ctrl-C on Windows
- PR #5991[1257] - Support IPV6 on command line for explicit network initialization
- PR #5990[1258] - P2300 enhancements
- PR #5989[1259] - Fix missing documentation in Public API page
- PR #5987[1260] - Attempting to fix timed executor API
- PR #5986[1261] - Fix warnings when building docs
- PR #5985[1262] - Re-add deprecated tag_policy_tag et.al. types that were removed in V1.8.1
- PR #5981[1263] - docs: add docs for condition_variable.hpp
- PR #5980[1264] - More work on execution::read
- PR #5979[1265] - Unsupport clang-v8 and clang-v9, switch LSU clang-v13 to C++17
- PR #5977[1266] - fix: Compilation errors for -std=c++17 builders
- PR #5975[1267] - docs: fix & improve parallel algorithms documentation 5
- PR #5974[1268] - [P2300] Adapt get completion signatures for awaitable senders
- PR #5973[1269] - defaults boost.context on riscv64
- PR #5972[1270] - Fix documentation for container algorithms
- PR #5971[1271] - added logic to detect riscv compiler configured for 64 bit target
- PR #5968[1272] - adds risc-v 64 bit support

[1250] https://github.com/STEllAR-GROUP/hpx/pull/6004
[1251] https://github.com/STEllAR-GROUP/hpx/pull/6003
[1252] https://github.com/STEllAR-GROUP/hpx/pull/6002
[1253] https://github.com/STEllAR-GROUP/hpx/pull/5996
[1254] https://github.com/STEllAR-GROUP/hpx/pull/5994
[1255] https://github.com/STEllAR-GROUP/hpx/pull/5993
[1256] https://github.com/STEllAR-GROUP/hpx/pull/5992
[1257] https://github.com/STEllAR-GROUP/hpx/pull/5991
[1258] https://github.com/STEllAR-GROUP/hpx/pull/5990
[1259] https://github.com/STEllAR-GROUP/hpx/pull/5989
[1260] https://github.com/STEllAR-GROUP/hpx/pull/5987
[1261] https://github.com/STEllAR-GROUP/hpx/pull/5986
[1262] https://github.com/STEllAR-GROUP/hpx/pull/5985
[1263] https://github.com/STEllAR-GROUP/hpx/pull/5981
[1264] https://github.com/STEllAR-GROUP/hpx/pull/5980
[1265] https://github.com/STEllAR-GROUP/hpx/pull/5979
[1266] https://github.com/STEllAR-GROUP/hpx/pull/5977
[1267] https://github.com/STEllAR-GROUP/hpx/pull/5975
[1268] https://github.com/STEllAR-GROUP/hpx/pull/5974
[1269] https://github.com/STEllAR-GROUP/hpx/pull/5973
[1270] https://github.com/STEllAR-GROUP/hpx/pull/5972
[1271] https://github.com/STEllAR-GROUP/hpx/pull/5971
[1272] https://github.com/STEllAR-GROUP/hpx/pull/5968

- PR #5967[1273] - Adding missing pieces to sync_wait, adding run_loop
- PR #5966[1274] - docs: fix & improve parallel algorithms documentation 4
- PR #5965[1275] - Fixing inspect problems, adding missing header file
- PR #5962[1276] - Changes in html page of documentation
- PR #5961[1277] - Prevent stalling during shutdown when running hello_world_distributed
- PR #5955[1278] - Fix documentation for container algorithms
- PR #5952[1279] - docs: fix & improve parallel algorithms documentation 3
- PR #5950[1280] - Change executors to directly implement the executor CPOs
- PR #5949[1281] - Converting async combinators into CPOs
- PR #5948[1282] - Adding support for pure sender/receiver based executors to parallel algorithms
- PR #5945[1283] - [P2300] Added fundamental coroutine_traits for S/R
- PR #5883[1284] - Optimization on LCI parcelport: uses LCI_putva
- PR #5872[1285] - Block fork join executor
- PR #5855[1286] - Adding performance test Jenkins builder at LSU

HPX V1.8.1 (Aug 5, 2022)

This is a bugfix release with a few minor additions and resolved problems.

General changes

This patch release adds a number of small new features and fixes a handful of problems discovered since the last release, in particular:

- A lot of work has been done to improve vectorization support for our parallel algorithms. HPX now supports using EVE - the Expressive Vector Engine as a vectorization backend.
- Added a simple average power consumption performance counter.
- Added performance counters related to the use of zero-copy chunks in the networking layer.
- More work was done towards full compatibility with the sender/receivers proposal P2300.
- Fixing sync_wait to decay the result types
- Fixed collective operations to properly avoid overalapping consecutive operations on the same communicator.

[1273] https://github.com/STEllAR-GROUP/hpx/pull/5967
[1274] https://github.com/STEllAR-GROUP/hpx/pull/5966
[1275] https://github.com/STEllAR-GROUP/hpx/pull/5965
[1276] https://github.com/STEllAR-GROUP/hpx/pull/5962
[1277] https://github.com/STEllAR-GROUP/hpx/pull/5961
[1278] https://github.com/STEllAR-GROUP/hpx/pull/5955
[1279] https://github.com/STEllAR-GROUP/hpx/pull/5952
[1280] https://github.com/STEllAR-GROUP/hpx/pull/5950
[1281] https://github.com/STEllAR-GROUP/hpx/pull/5949
[1282] https://github.com/STEllAR-GROUP/hpx/pull/5948
[1283] https://github.com/STEllAR-GROUP/hpx/pull/5945
[1284] https://github.com/STEllAR-GROUP/hpx/pull/5883
[1285] https://github.com/STEllAR-GROUP/hpx/pull/5872
[1286] https://github.com/STEllAR-GROUP/hpx/pull/5855

- Simplified the implementation of our execution policies and added mapping functions between those.
- Fixed performance issues with our implementation of *small_vector*.
- Serialization now works with buffers of unsigned characters.
- Fixing dangling reference in serialization of non-default constructible types
- Fixed static linking on Windows.
- Fixed support for M1/MacOS based architectures.
- Fixed support for gentoo/musl.
- Fixed *hpx::counting_semaphore_var*.
- Properly check start and end bounds for *hpx::for_loop*
- A lot of changes and fixes to the documentation (see https://hpx-docs.stellar-group.org).

Breaking changes

- No breaking changes have been introduced.

Closed issues

- Issue #5964[1287] - component with multiple inheritance
- Issue #5946[1288] - dll_dlopen.hpp: error: RTLD_DI_ORIGIN was not declared in this scope with musl libc
- Issue #5925[1289] - Simplify implementation of execution policies
- Issue #5924[1290] - {what}: mmap() failed to allocate thread stack: HPX(unhandled_exception)
- Issue #5912[1291] - collectives all gather hangs if rank 0 is not involved
- Issue #5902[1292] - MPI parcelport issue on Fugaku
- Issue #5900[1293] - Unable to build hello_world_distributed.cpp.
- Issue #5892[1294] - Problems with HPX serialization as a standalone feature. Testcase provided.
- Issue #5886[1295] - Segfault when serializing non default constructible class with stl containers data members
- Issue #5832[1296] - Distributed execution crash
- Issue #5768[1297] - HPX hangs on Perlmutter
- Issue #5735[1298] - hpx::for_loop executes without checking start and end bounds
- Issue #5700[1299] - HPX(serialization_error)

1287 https://github.com/STEllAR-GROUP/hpx/issues/5964
1288 https://github.com/STEllAR-GROUP/hpx/issues/5946
1289 https://github.com/STEllAR-GROUP/hpx/issues/5925
1290 https://github.com/STEllAR-GROUP/hpx/issues/5924
1291 https://github.com/STEllAR-GROUP/hpx/issues/5912
1292 https://github.com/STEllAR-GROUP/hpx/issues/5902
1293 https://github.com/STEllAR-GROUP/hpx/issues/5900
1294 https://github.com/STEllAR-GROUP/hpx/issues/5892
1295 https://github.com/STEllAR-GROUP/hpx/issues/5886
1296 https://github.com/STEllAR-GROUP/hpx/issues/5832
1297 https://github.com/STEllAR-GROUP/hpx/issues/5768
1298 https://github.com/STEllAR-GROUP/hpx/issues/5735
1299 https://github.com/STEllAR-GROUP/hpx/issues/5700

Closed pull requests

- PR #5970[1300] - Fixing component multiple inheritance
- PR #5969[1301] - Fixing sync_wait to avoid dangling references
- PR #5963[1302] - Fixing sync_wait to decay the result types
- PR #5960[1303] - docs: added name to documentation contributors list
- PR #5959[1304] - Fixing sync_wait to decay the result types
- PR #5954[1305] - refactor: rename itr to correct type (*reduce*)
- PR #5954[1306] - refactor: rename itr to correct type (*reduce*)
- PR #5953[1307] - Fixed property handling in hierarchical_spawning
- PR #5951[1308] - Fixing static linking (for Windows)
- PR #5947[1309] - Fix building on musl.
- PR #5944[1310] - added adaptive_static_chunk_size
- PR #5943[1311] - Fix sync_wait
- PR #5942[1312] - Fix doc warnings
- PR #5941[1313] - Fix sync_wait
- PR #5940[1314] - Protect collective operations against std::vector<bool> idiosyncrasies
- PR #5939[1315] - docs: fix & improve parallel algorithms documentation 2
- PR #5938[1316] - Properly implement generation support for collective operations
- PR #5937[1317] - Remove leftover files from PMR based small_vector
- PR #5936[1318] - Adding mapping functions between execution policies
- PR #5935[1319] - Fixing serialization to work with buffers of unsigned chars
- PR #5934[1320] - Attempting to fix datapar issues on CircleCI
- PR #5933[1321] - Fix documentation for ranges algorithms

[1300] https://github.com/STEllAR-GROUP/hpx/pull/5970
[1301] https://github.com/STEllAR-GROUP/hpx/pull/5969
[1302] https://github.com/STEllAR-GROUP/hpx/pull/5963
[1303] https://github.com/STEllAR-GROUP/hpx/pull/5960
[1304] https://github.com/STEllAR-GROUP/hpx/pull/5959
[1305] https://github.com/STEllAR-GROUP/hpx/pull/5954
[1306] https://github.com/STEllAR-GROUP/hpx/pull/5954
[1307] https://github.com/STEllAR-GROUP/hpx/pull/5953
[1308] https://github.com/STEllAR-GROUP/hpx/pull/5951
[1309] https://github.com/STEllAR-GROUP/hpx/pull/5947
[1310] https://github.com/STEllAR-GROUP/hpx/pull/5944
[1311] https://github.com/STEllAR-GROUP/hpx/pull/5943
[1312] https://github.com/STEllAR-GROUP/hpx/pull/5942
[1313] https://github.com/STEllAR-GROUP/hpx/pull/5941
[1314] https://github.com/STEllAR-GROUP/hpx/pull/5940
[1315] https://github.com/STEllAR-GROUP/hpx/pull/5939
[1316] https://github.com/STEllAR-GROUP/hpx/pull/5938
[1317] https://github.com/STEllAR-GROUP/hpx/pull/5937
[1318] https://github.com/STEllAR-GROUP/hpx/pull/5936
[1319] https://github.com/STEllAR-GROUP/hpx/pull/5935
[1320] https://github.com/STEllAR-GROUP/hpx/pull/5934
[1321] https://github.com/STEllAR-GROUP/hpx/pull/5933

- PR #5932[1322] - Remove mimalloc version constraint
- PR #5931[1323] - docs: fix & improve parallel algorithms documentation
- PR #5930[1324] - Add boost to hip builder
- PR #5929[1325] - Apply fixes to M1/MacOS related stack allocation to all relevant spots
- PR #5928[1326] - updated context_generic_context to accommodate arm64_arch_8/Apple architecture
- PR #5927[1327] - Public derivation for counting_semaphore_var
- PR #5926[1328] - Fix doxygen warnings when building documentation
- PR #5923[1329] - Fixing git checkout to reflect latest version tag
- PR #5922[1330] - A couple of unrelated changes in support of implementing P1673
- PR #5920[1331] - [P2300] enhancements: receiver_of, sender_of improvements
- PR #5917[1332] - Fixing various 'held lock while suspending' problems
- PR #5916[1333] - Fix minor doxygen parsing typo
- PR #5915[1334] - docs: fix broken api algo links
- PR #5914[1335] - Remove CSS rules - update sphinx version
- PR #5911[1336] - Removed references to hpx::vector in comments
- PR #5909[1337] - Remove stuff which is defined in the header
- PR #5906[1338] - Use BUILD_SHARED_LIBS correctly
- PR #5905[1339] - Fix incorrect usage of generator expressions
- PR #5904[1340] - Delete FindBZip2.cmake
- PR #5901[1341] - Fix #5900
- PR #5899[1342] - Replace PMR based version of small_vector
- PR #5897[1343] - Add missing ""
- PR #5896[1344] - Docs: Add serialization tutorial.

[1322] https://github.com/STEllAR-GROUP/hpx/pull/5932
[1323] https://github.com/STEllAR-GROUP/hpx/pull/5931
[1324] https://github.com/STEllAR-GROUP/hpx/pull/5930
[1325] https://github.com/STEllAR-GROUP/hpx/pull/5929
[1326] https://github.com/STEllAR-GROUP/hpx/pull/5928
[1327] https://github.com/STEllAR-GROUP/hpx/pull/5927
[1328] https://github.com/STEllAR-GROUP/hpx/pull/5926
[1329] https://github.com/STEllAR-GROUP/hpx/pull/5923
[1330] https://github.com/STEllAR-GROUP/hpx/pull/5922
[1331] https://github.com/STEllAR-GROUP/hpx/pull/5920
[1332] https://github.com/STEllAR-GROUP/hpx/pull/5917
[1333] https://github.com/STEllAR-GROUP/hpx/pull/5916
[1334] https://github.com/STEllAR-GROUP/hpx/pull/5915
[1335] https://github.com/STEllAR-GROUP/hpx/pull/5914
[1336] https://github.com/STEllAR-GROUP/hpx/pull/5911
[1337] https://github.com/STEllAR-GROUP/hpx/pull/5909
[1338] https://github.com/STEllAR-GROUP/hpx/pull/5906
[1339] https://github.com/STEllAR-GROUP/hpx/pull/5905
[1340] https://github.com/STEllAR-GROUP/hpx/pull/5904
[1341] https://github.com/STEllAR-GROUP/hpx/pull/5901
[1342] https://github.com/STEllAR-GROUP/hpx/pull/5899
[1343] https://github.com/STEllAR-GROUP/hpx/pull/5897
[1344] https://github.com/STEllAR-GROUP/hpx/pull/5896

- PR #5895[1345] - Update to V1.9.0 on master
- PR #5894[1346] - Fix executor_with_thread_hooks example
- PR #5890[1347] - Adding simple average power consumption performance counter
- PR #5889[1348] - Par unseq/unseq adding
- PR #5888[1349] - Support for data-parallelism for reduce, transform reduce, transform_binary_reduce algorithms
- PR #5887[1350] - Fixing dangling reference in serialization of non-default constructible types
- PR #5879[1351] - New performance counters related to zero-copy chunks.

HPX V1.8.0 (May 18, 2022)

With HPX parallel algorithms been fully adapted to C++20 the new release achieves full conformance with C++20 concurrency and parallelism facilities. HPX now supports all of the algorithms as specified by C++20. We have added support for vectorization to more of our algorithms. Much work has been done towards implementing P2300 ("std::execution") and implementing the underlying senders/receivers facilities. Finally, The new release comes with a brand new documentation interface!

General changes

- The new documentation can now be found on our webpage: https://hpx-docs.stellar-group.org. This includes a completely new and user-friendly interface environment along with restructuring of certain components. The content in the "Quick start", "Manual" and "Examples" was improved, while the "Build system" page was adapted to include necessary information for newcommers.
- With the vectorization support available in modern hardware architectures HPX now provides new data-parallel vector execution policies `hpx::execution::simd` and `hpx::execution::par_simd` that enable significant speed-up of our parallel algorithm implementations. The following algorithms now support SIMD execution:
  - `copy`, `copy_n`
  - `generate`
  - `adjacent_difference`, `adjacent_find`
  - `all_of`, `any_of`, `none_of`
  - `equal`, `mismatch`,
  - `inner_product`
  - `count`, `count_if`
  - `fill`, `fill_n`
  - `find`, `find_end`, `find_first_of`, `find_if`, `find_if_not`
  - `for_each`, `for_each_n`
  - `generate`, `generate_n`.

[1345] https://github.com/STEllAR-GROUP/hpx/pull/5895
[1346] https://github.com/STEllAR-GROUP/hpx/pull/5894
[1347] https://github.com/STEllAR-GROUP/hpx/pull/5890
[1348] https://github.com/STEllAR-GROUP/hpx/pull/5889
[1349] https://github.com/STEllAR-GROUP/hpx/pull/5888
[1350] https://github.com/STEllAR-GROUP/hpx/pull/5887
[1351] https://github.com/STEllAR-GROUP/hpx/pull/5879

- Based on top of P2300 the HPX parallel algorithms now support the pipeline syntax towards an effort to unify their usage along with senders/receivers. The HPX parallel algorithms can now bind with senders/receivers using the pipeline operator.
- Several changes took place on the executors provided by HPX:
- The executors now support the `num_cores` options in order for the user to be able to specify the desired number of cores to be used in the correspodning execution.
- The `scheduler` executor was implemented on top of senders/receivers and can be used with all HPX facilities that schedule new work, such as parallel algorithms, `hpx::async`, `hpx::dataflow`, etc.
- The performance of `fork_join_executor` was improved.
- The following algorithms have been added/adapted to be C++20 conformant:
  - `min_element`
  - `max_element`
  - `minmax_element`
  - `starts_with`
  - `ends_with`
  - `swap_ranges`
  - `unique`
  - `unique_copy`
  - `rotate`
  - `rotate_copy`
  - `sort`
  - `shift_left`
  - `shift_right`
  - `stable_sort`
  - `partition`
  - `partition_copy`
  - `stable_partition`
  - `adjacent_difference`
  - `nth_element`
  - `partial_sort`
  - `partial_sort_copy`.
- `HPX_FORWARD`/`HPX_MOVE` macros were introduced that replaced the `std::move` and `std::forward` facilities that in the library code.
- Hangs on distributed barrier were fixed.
- The performance of `scan_partitioner` was improved.
- Support was added for `thread_priority` to the `parallel_execution_policy`

- Regarding senders/receivers and the P2300 proposal various actions took place. stop_token was adapted to the recent proposal version (in_place_stop_token was introduced). Also hint, annotation, priority and stacksize properties were added to the scheduler executor. Stop support was added to when_all. Support for completion signatures was added. The following schedulers and algorithms were added:
  - get_completion_scheduler
  - any_sender and unique_any_sender
  - split sender
  - transform_mpi sender
  - transfer sender
  - let_error, let_stopped
  - get_env and related environment queries
  - schedule, set_value, set_error, set_done, start and connect are now proper customization points as defined in P2300.
- Several namespaces were altered towards conformance with C++20. Compatibility layers have been added and the old versions will be removed in next releases. The namespace changes are the following:
  - hpx::parallel::induction/reduction were movied into namespace hpx::experimental
  - for_loop and friends were moved into namespace hpx::experimental.
  - hpx::util::optional and friends were moved into namespace hpx.
  - hpx::lcos::barrier has been moved into the hpx::distributed namespace and hpx::lcos::local::cpp20_barrier has been renamed to barrier and moved into the hpx namespace.
  - hpx::lcos::latch has been moved into the hpx::distributed namespace and lcos::local::latch has been moved into the hpx namespace. The count_down_and_wait() functionality of latch has been renamed to arrive_and_wait().
  - hpx::util::unique_function_nonser has been renamed to hpx::move_only_function.
  - hpx::util::unique_function has been renamed to hpx::distributed::move_only_function.
  - hpx::util::function has been renamed to hpx::distributed::function.
  - hpx::util::function_nonser has been renamed to hpx::function.
  - hpx::util::function_ref have been moved to namespace hpx.
  - hpx::lcos::split_future changed namespace and is now used as hpx::split_future.
  - hpx::lcos::local::counting_semaphore has been deprecated and hpx::lcos::local::cpp20_counting_semaphore has been renamed to hpx::counting_semaphore.
  - hpx::lcos::local::cpp20_binary_semaphore has been renamed to hpx::binary_semaphore.
  - hpx::lcos::local::sliding_semaphore has been renamed to hpx::sliding_semaphore and
  - hpx::lcos::local::sliding_semaphore_var has been renamed to hpx::sliding_semaphore_var.
  - hpx::lcos::local::spinlock has been renamed to hpx::spinlock.
  - hpx::lcos::local::mutex has been renamed to hpx::mutex.
  - hpx::lcos::local::timed_mutex has been renamed to hpx::timed_mutex.

- hpx::lcos::local::no_mutex has been renamed to hpx::no_mutex.
- hpx::lcos::local::recursive_mutex has been renamed to hpx::recursive_mutex.
- hpx::lcos::local::shared_mutex has been renamed to hpx::shared_mutex.
- hpx::lcos::local::upgrade_lock has been renamed to hpx::upgrade_lock.
- hpx::lcos::local::upgrade_to_unique_lock has been renamed to hpx::upgrade_to_unique_lock.
- hpx::lcos::local::condition_variable has been renamed to hpx::condition_variable. hpx::lcos::local::condition_variable_var has been renamed to hpx::condition_variable_var.
- hpx::lcos::local::once_flag has been renamed to hpx::once_flag, and . hpx::lcos::local::call_once has been renamed to hpx::call_once.

- The new LCI (Lightweight Communication Interface) parcelport was added that supports irregular and asynchronous applications like graph analysis, sparce linear algebra, modern parallel architectures etc. Major features include:
  - Support for advanced communication primitives like two sided send/recv and one sided remote put.
  - Better multi-threaded performance.
  - Explicit user control of communication resource.
  - Flexible signaling mechanisms (synchronizer, completion queue, active message handler).
- The following CMake flags were added, mostly to support using HPX as a backend for SHAD (https://github.com/pnnl/SHAD). Please note that these options enable questionable functionalities, partially they even enable undefined behavior. Please only use any of them if you know what you're doing:
  - HPX_SERIALIZATION_WITH_ALLOW_RAW_POINTER_SERIALIZATION
  - HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE
  - HPX_SERIALIZATION_WITH_ALLOW_CONST_TUPLE_MEMBERS

Breaking changes

- Minimum required C++ standard library is C++17.
- Support for GCC 7 and Clang 8.0.0 and below has been removed.
- CUDA version required updated to 11.4.
- CMake version required updated to 3.18.
- The default version of Asio used was updated to 1.20.0.
- The default version of APEX used was updated to 2.5.1.
- APEX version was updated to 2.5.1.
- tagged_pair and tagged_tuple were removed.
- tag_dispatch was renamed to tag_invoke.
- hpx.max_backgroud_threads was renamed to hpx.parcel.max_background_threads.
- The following CMake flags were removed after being deprecated for at least two releases:
  - HPX_SCHEDULER_MAX_TERMINATED_THREADS

    - HPX_WITH_GOOGLE_PERFTOOLS
    - HPX_WITH_INIT_START_OVERLOADS_COMPATIBILITY
    - HPX_HAVE_{COROUTINE,PLUGIN}_GCC_HIDDEN_VISIBILITY
    - HPX_TOP_LEVEL
    - HPX_WITH_COMPUTE_CUDA
    - HPX_WITH_ASYNC_CUDA

- `annotate_function` was renamed to `scoped_annotation`.
- `execution::transform` was renamed to `execution::then`.
- `execution::detach` was renamed to `execution::start_detached`.
- `execution::on_sender` was renamed to `execution::schedule_on`.
- `execution::just_on` was renamed to `execution::just_transfer`.
- `execution::set_done` was renamed to `execution::set_stopped`.

Closed issues

- Issue #5871[1352] - distributed::channel.regsiter_as terminates the active task.
- Issue #5856[1353] - Performance counters do not compile
- Issue #5828[1354] - hpx::distributed:barrier errors
- Issue #5812[1355] - OctoTiger does not compile with HPX master and CUDA 11.5
- Issue #5784[1356] - HPX failing with co_await and hpx::when_all(futures)
- Issue #5774[1357] - CMake can't find HPXCacheVariables.cmake
- Issue #5764[1358] - Fix HIP problem
- Issue #5724[1359] - Missing binary filter compression header
- Issue #5721[1360] - Cleanup after repository split
- Issue #5701[1361] - It seems that the tcp parcelport is running, and the MPI parcelport is ignored
- Issue #5692[1362] - Kokkos compilation fails when using both HPX and CUDA execution spaces with gcc 9.3.0
- Issue #5686[1363] - Rename *annotate_function*
- Issue #5668[1364] - HPX does not detect the C++ 20 standard using gcc 11.2
- Issue #5666[1365] - Compilation error using boost 1.76 and gcc 11.2.1

[1352] https://github.com/STEllAR-GROUP/hpx/issues/5871
[1353] https://github.com/STEllAR-GROUP/hpx/issues/5856
[1354] https://github.com/STEllAR-GROUP/hpx/issues/5828
[1355] https://github.com/STEllAR-GROUP/hpx/issues/5812
[1356] https://github.com/STEllAR-GROUP/hpx/issues/5784
[1357] https://github.com/STEllAR-GROUP/hpx/issues/5774
[1358] https://github.com/STEllAR-GROUP/hpx/issues/5764
[1359] https://github.com/STEllAR-GROUP/hpx/issues/5724
[1360] https://github.com/STEllAR-GROUP/hpx/issues/5721
[1361] https://github.com/STEllAR-GROUP/hpx/issues/5701
[1362] https://github.com/STEllAR-GROUP/hpx/issues/5692
[1363] https://github.com/STEllAR-GROUP/hpx/issues/5686
[1364] https://github.com/STEllAR-GROUP/hpx/issues/5668
[1365] https://github.com/STEllAR-GROUP/hpx/issues/5666

- Issue #5653[1366] - Implement P2248 for our algorithms
- Issue #5647[1367] - [User input needed] Remove (CUDA) compute functionality?
- Issue #5590[1368] - hello_world_distributed fails on startup with HPX stable, MPICH 3.3.2, on Deep Bayou
- Issue #5570[1369] - Rename tag_dispatch to tag_invoke
- Issue #5566[1370] - can't build simple example: "Cannot use the dummy implementation of future_then_dispatch"
- Issue #5565[1371] - build failure: hpx::string_util::trim()
- Issue #5553[1372] - Github action to validate the cff file refs #5471
- Issue #5504[1373] - CMake does not work for HPX 1.7.0 on Piz Daint
- Issue #5503[1374] - Use contiguous index queue in bulk execution to reduce number of spawned tasks
- Issue #5502[1375] - C++20 std::coroutine cmake detection
- Issue #5478[1376] - hpx.dll built with vcpkg got functions pointing to the same location
- Issue #5472[1377] - Compilation error with cuda/11.3
- Issue #5469[1378] - Compiler warning about HPX_NODISCARD when building with APEX
- Issue #5463[1379] - Address minor comments of the C++17 PR bump
- Issue #5456[1380] - Use *std::ranges::iter_swap* where available
- Issue #5404[1381] - Build fails with error "Cannot open include file asio/io_context.hpp"
- Issue #5381[1382] - Add starts_with and ends_with algorithms
- Issue #5344[1383] - Further simplify tag_invoke helpers
- Issue #5269[1384] - Allow setting a label on executors/policies
- Issue #5219[1385] - (Re-)Implement executor API on top of sender/receiver infrastructure
- Issue #5216[1386] - Performance counter module not loading
- Issue #5162[1387] - Require C++17 support
- Issue #5156[1388] - Disentangle segmented algorithms

[1366] https://github.com/STEllAR-GROUP/hpx/issues/5653
[1367] https://github.com/STEllAR-GROUP/hpx/issues/5647
[1368] https://github.com/STEllAR-GROUP/hpx/issues/5590
[1369] https://github.com/STEllAR-GROUP/hpx/issues/5570
[1370] https://github.com/STEllAR-GROUP/hpx/issues/5566
[1371] https://github.com/STEllAR-GROUP/hpx/issues/5565
[1372] https://github.com/STEllAR-GROUP/hpx/issues/5553
[1373] https://github.com/STEllAR-GROUP/hpx/issues/5504
[1374] https://github.com/STEllAR-GROUP/hpx/issues/5503
[1375] https://github.com/STEllAR-GROUP/hpx/issues/5502
[1376] https://github.com/STEllAR-GROUP/hpx/issues/5478
[1377] https://github.com/STEllAR-GROUP/hpx/issues/5472
[1378] https://github.com/STEllAR-GROUP/hpx/issues/5469
[1379] https://github.com/STEllAR-GROUP/hpx/issues/5463
[1380] https://github.com/STEllAR-GROUP/hpx/issues/5456
[1381] https://github.com/STEllAR-GROUP/hpx/issues/5404
[1382] https://github.com/STEllAR-GROUP/hpx/issues/5381
[1383] https://github.com/STEllAR-GROUP/hpx/issues/5344
[1384] https://github.com/STEllAR-GROUP/hpx/issues/5269
[1385] https://github.com/STEllAR-GROUP/hpx/issues/5219
[1386] https://github.com/STEllAR-GROUP/hpx/issues/5216
[1387] https://github.com/STEllAR-GROUP/hpx/issues/5162
[1388] https://github.com/STEllAR-GROUP/hpx/issues/5156

- Issue #5118[1389] - Lock held while suspending
- Issue #5111[1390] - Tests fail to build with binary_filter plugins enabled
- Issue #5110[1391] - Tests don't get built
- Issue #5105[1392] - PAPI performance counters not available
- Issue #5002[1393] - hpx::lcos::barrier() results in deadlock
- Issue #4992[1394] - Clang-format the rest of the files
- Issue #4987[1395] - Use std::function in public APIs
- Issue #4871[1396] - HEP: conformance to C++20
- Issue #4822[1397] - Adapt parallel algorithms to C++20
- Issue #4736[1398] - Deprecate hpx::flush and hpx::endl
- Issue #4558[1399] - Prevent work-stealing from stalling
- Issue #4495[1400] - Add anchor links to table rows in documentation
- Issue #4469[1401] - New thread state: *pending_low*
- Issue #4321[1402] - After the modularization the libfabric parcelport does not compile
- Issue #4308[1403] - Using APEX on multinode jobs when HPX_WITH_NETWORKING = OFF
- Issue #3995[1404] - Use C++20 std::source_location where available, adapt ours to conform
- Issue #3861[1405] - Selected processor does not support 'yield' in ARM mode
- Issue #3706[1406] - Add shift_left and shift_right algorithms
- Issue #3646[1407] - Parallel algorithms should accept iterator/sentinel pairs
- Issue #3636[1408] - HPX Modularization
- Issue #3546[1409] - Modularization of HPX
- Issue #3474[1410] - Modernize CMake used in HPX
- Issue #1836[1411] - hpx::parallel does not have a sort implementation

[1389] https://github.com/STEllAR-GROUP/hpx/issues/5118
[1390] https://github.com/STEllAR-GROUP/hpx/issues/5111
[1391] https://github.com/STEllAR-GROUP/hpx/issues/5110
[1392] https://github.com/STEllAR-GROUP/hpx/issues/5105
[1393] https://github.com/STEllAR-GROUP/hpx/issues/5002
[1394] https://github.com/STEllAR-GROUP/hpx/issues/4992
[1395] https://github.com/STEllAR-GROUP/hpx/issues/4987
[1396] https://github.com/STEllAR-GROUP/hpx/issues/4871
[1397] https://github.com/STEllAR-GROUP/hpx/issues/4822
[1398] https://github.com/STEllAR-GROUP/hpx/issues/4736
[1399] https://github.com/STEllAR-GROUP/hpx/issues/4558
[1400] https://github.com/STEllAR-GROUP/hpx/issues/4495
[1401] https://github.com/STEllAR-GROUP/hpx/issues/4469
[1402] https://github.com/STEllAR-GROUP/hpx/issues/4321
[1403] https://github.com/STEllAR-GROUP/hpx/issues/4308
[1404] https://github.com/STEllAR-GROUP/hpx/issues/3995
[1405] https://github.com/STEllAR-GROUP/hpx/issues/3861
[1406] https://github.com/STEllAR-GROUP/hpx/issues/3706
[1407] https://github.com/STEllAR-GROUP/hpx/issues/3646
[1408] https://github.com/STEllAR-GROUP/hpx/issues/3636
[1409] https://github.com/STEllAR-GROUP/hpx/issues/3546
[1410] https://github.com/STEllAR-GROUP/hpx/issues/3474
[1411] https://github.com/STEllAR-GROUP/hpx/issues/1836

- Issue #1668[1412] - Adapt all parallel algorithms to Ranges TS
- Issue #1141[1413] - Implement N4409 on top of HPX

Closed pull requests

- PR #5885[1414] - Testing newer ASIO version
- PR #5884[1415] - Fix miscellaneous doc sections
- PR #5882[1416] - Fixing OctoTiger incompatibility introduced recently
- PR #5881[1417] - Fixing recent patch that disables ATOMIC_FLAG_INIT for C++20 and up
- PR #5880[1418] - refactor: convert *counter_status* enum to enum class
- PR #5878[1419] - Docs: Replaced non-existent create_reducer function with create_communicator
- PR #5877[1420] - Doc updates hpx runtime and resources
- PR #5876[1421] - Updates to documentation; grammar edits.
- PR #5875[1422] - Doc updates starting the hpx runtime
- PR #5874[1423] - Doc updates launching configuring
- PR #5873[1424] - Prevent certain generated files from being deleted on reconfigure
- PR #5870[1425] - Adding support for the PJM batch environment
- PR #5867[1426] - Update CMakeLists.txt
- PR #5866[1427] - add cmake option HPX_WITH_PARCELPORT_COUNTERS
- PR #5864[1428] - ATOMIC_INIT_FLAG is deprecated starting C++20
- PR #5863[1429] - Adding llvm 14.0.0 with boost 1.79.0 to Jenkins
- PR #5861[1430] - Let install step proceed on CircleCI even if the segmented algorithms fail
- PR #5860[1431] - Updating APEX tag
- PR #5859[1432] - Splitting documentation generation steps on CircleCI
- PR #5854[1433] - Fixing left-overs from changing counter_type to enum class

[1412] https://github.com/STEllAR-GROUP/hpx/issues/1668
[1413] https://github.com/STEllAR-GROUP/hpx/issues/1141
[1414] https://github.com/STEllAR-GROUP/hpx/pull/5885
[1415] https://github.com/STEllAR-GROUP/hpx/pull/5884
[1416] https://github.com/STEllAR-GROUP/hpx/pull/5882
[1417] https://github.com/STEllAR-GROUP/hpx/pull/5881
[1418] https://github.com/STEllAR-GROUP/hpx/pull/5880
[1419] https://github.com/STEllAR-GROUP/hpx/pull/5878
[1420] https://github.com/STEllAR-GROUP/hpx/pull/5877
[1421] https://github.com/STEllAR-GROUP/hpx/pull/5876
[1422] https://github.com/STEllAR-GROUP/hpx/pull/5875
[1423] https://github.com/STEllAR-GROUP/hpx/pull/5874
[1424] https://github.com/STEllAR-GROUP/hpx/pull/5873
[1425] https://github.com/STEllAR-GROUP/hpx/pull/5870
[1426] https://github.com/STEllAR-GROUP/hpx/pull/5867
[1427] https://github.com/STEllAR-GROUP/hpx/pull/5866
[1428] https://github.com/STEllAR-GROUP/hpx/pull/5864
[1429] https://github.com/STEllAR-GROUP/hpx/pull/5863
[1430] https://github.com/STEllAR-GROUP/hpx/pull/5861
[1431] https://github.com/STEllAR-GROUP/hpx/pull/5860
[1432] https://github.com/STEllAR-GROUP/hpx/pull/5859
[1433] https://github.com/STEllAR-GROUP/hpx/pull/5854

- PR #5853[1434] - Adding HPX dependency tool (adapted from Boostdep tool)
- PR #5852[1435] - Optimize LCI parcelport
- PR #5851[1436] - Forking dynamic_bitset from Boost
- PR #5850[1437] - Convert perf_counters::counter_type enum to enum class.
- PR #5849[1438] - Update LCI parcelport to LCI v1.7.1
- PR #5848[1439] - Fedora related fixes
- PR #5847[1440] - Fix API, troubleshooting & people
- PR #5844[1441] - Attempting to fix timeouts of segmented iterator tests
- PR #5842[1442] - change the default value of HPX_WITH_LCI_TAG to v1.7
- PR #5841[1443] - Move the split_future facilities into the namespace hpx
- PR #5840[1444] - wait_xxx_nothrow functions return whether one of the futures is exceptional
- PR #5839[1445] - Moving a list of synchronization primitives into namespace hpx
- PR #5837[1446] - Moving latch types to hpx and hpx::distributed namespaces
- PR #5835[1447] - Add missing compatibility layer for id_type::management_type values
- PR #5834[1448] - API docs changes
- PR #5831[1449] - Further improvement actions to rotate
- PR #5830[1450] - Exposing zero-copy serialization threshold through configuration option
- PR #5829[1451] - Attempting to fix failing barrier test
- PR #5827[1452] - Add back explicit template parameter to *ignore_while_checking* to compile with nvcc
- PR #5826[1453] - Reduce number of allocations while calling async_bulk_execute
- PR #5825[1454] - Steal from neighboring NUMA domain only
- PR #5823[1455] - Remove obsolete directories and adjust build system
- PR #5822[1456] - Clang-format remaining files

[1434] https://github.com/STEllAR-GROUP/hpx/pull/5853
[1435] https://github.com/STEllAR-GROUP/hpx/pull/5852
[1436] https://github.com/STEllAR-GROUP/hpx/pull/5851
[1437] https://github.com/STEllAR-GROUP/hpx/pull/5850
[1438] https://github.com/STEllAR-GROUP/hpx/pull/5849
[1439] https://github.com/STEllAR-GROUP/hpx/pull/5848
[1440] https://github.com/STEllAR-GROUP/hpx/pull/5847
[1441] https://github.com/STEllAR-GROUP/hpx/pull/5844
[1442] https://github.com/STEllAR-GROUP/hpx/pull/5842
[1443] https://github.com/STEllAR-GROUP/hpx/pull/5841
[1444] https://github.com/STEllAR-GROUP/hpx/pull/5840
[1445] https://github.com/STEllAR-GROUP/hpx/pull/5839
[1446] https://github.com/STEllAR-GROUP/hpx/pull/5837
[1447] https://github.com/STEllAR-GROUP/hpx/pull/5835
[1448] https://github.com/STEllAR-GROUP/hpx/pull/5834
[1449] https://github.com/STEllAR-GROUP/hpx/pull/5831
[1450] https://github.com/STEllAR-GROUP/hpx/pull/5830
[1451] https://github.com/STEllAR-GROUP/hpx/pull/5829
[1452] https://github.com/STEllAR-GROUP/hpx/pull/5827
[1453] https://github.com/STEllAR-GROUP/hpx/pull/5826
[1454] https://github.com/STEllAR-GROUP/hpx/pull/5825
[1455] https://github.com/STEllAR-GROUP/hpx/pull/5823
[1456] https://github.com/STEllAR-GROUP/hpx/pull/5822

- PR #5821[1457] - Enable permissive- flag on Windows GitHub actions builders
- PR #5820[1458] - Convert throwmode enum to enum class
- PR #5819[1459] - Marking customization points for intrusive_ptr as noexcept
- PR #5818[1460] - Unconditionally use C++17 attributes
- PR #5817[1461] - Modernize naming modules
- PR #5816[1462] - Modernize cache module
- PR #5815[1463] - Reapply flyby changes from #5467
- PR #5814[1464] - Avoid test timeouts by reducing test sizes
- PR #5813[1465] - The CUDA problem is not fixed in V11.5 yet…
- PR #5811[1466] - Make sure reduction value is properly moved, when possible
- PR #5810[1467] - Improve error reporting during device initialization in HIP environments
- PR #5809[1468] - Converting scheduler enums into enum class
- PR #5808[1469] - Deprecate hpx::flush and friends
- PR #5807[1470] - Use C++20 std::source_location, if available
- PR #5806[1471] - Moving promise and packaged_task to new namespaces
- PR #5805[1472] - Attempting to fix a test failure when using the LCI parcelpor
- PR #5803[1473] - Attempt to fix CUDA related OctoTiger problems
- PR #5800[1474] - Add option to restrict MPI background work to subset of cores
- PR #5798[1475] - Adding MPI as a dependency to APEX
- PR #5797[1476] - Extend Sphinx role to support arbitrary text to display on a link
- PR #5796[1477] - Disable CUDA tests that cause NVCC to silently fail without error messages
- PR #5795[1478] - Avoid writing path and directories into HPXCacheVariables.cmake
- PR #5793[1479] - Remove features that are deprecated since V1.6

1457 https://github.com/STEllAR-GROUP/hpx/pull/5821
1458 https://github.com/STEllAR-GROUP/hpx/pull/5820
1459 https://github.com/STEllAR-GROUP/hpx/pull/5819
1460 https://github.com/STEllAR-GROUP/hpx/pull/5818
1461 https://github.com/STEllAR-GROUP/hpx/pull/5817
1462 https://github.com/STEllAR-GROUP/hpx/pull/5816
1463 https://github.com/STEllAR-GROUP/hpx/pull/5815
1464 https://github.com/STEllAR-GROUP/hpx/pull/5814
1465 https://github.com/STEllAR-GROUP/hpx/pull/5813
1466 https://github.com/STEllAR-GROUP/hpx/pull/5811
1467 https://github.com/STEllAR-GROUP/hpx/pull/5810
1468 https://github.com/STEllAR-GROUP/hpx/pull/5809
1469 https://github.com/STEllAR-GROUP/hpx/pull/5808
1470 https://github.com/STEllAR-GROUP/hpx/pull/5807
1471 https://github.com/STEllAR-GROUP/hpx/pull/5806
1472 https://github.com/STEllAR-GROUP/hpx/pull/5805
1473 https://github.com/STEllAR-GROUP/hpx/pull/5803
1474 https://github.com/STEllAR-GROUP/hpx/pull/5800
1475 https://github.com/STEllAR-GROUP/hpx/pull/5798
1476 https://github.com/STEllAR-GROUP/hpx/pull/5797
1477 https://github.com/STEllAR-GROUP/hpx/pull/5796
1478 https://github.com/STEllAR-GROUP/hpx/pull/5795
1479 https://github.com/STEllAR-GROUP/hpx/pull/5793

- PR #5792[1480] - Making sure num_cores is properly handled by parallel_executor
- PR #5791[1481] - Moving bind, bind_front, bind_back to namespace hpx
- PR #5790[1482] - Moving serializable function/move_only_function into namespace hpx::distributed
- PR #5787[1483] - Remove unneeded (and commented) tests
- PR #5786[1484] - Attempting to fix hangs in distributed barrier
- PR #5785[1485] - add cmake code to detect arm64 on macOS
- PR #5783[1486] - Moving function and function_ref into namespace hpx
- PR #5781[1487] - Updating used version of Visual Studio
- PR #5780[1488] - Update Piz Daint Jenkins configurations from gcc/clang 7 to 8
- PR #5778[1489] - Updated for_loop.hpp
- PR #5777[1490] - Update reference for foreach benchmark
- PR #5775[1491] - Move optional into namespace hpx
- PR #5773[1492] - Moving barrier to consolidated namespaces
- PR #5772[1493] - Adding missing docs for ranges::find_if and find_if_not algorithms
- PR #5771[1494] - Moving for_loop into namespace hpx::experimental
- PR #5770[1495] - Fixing HIP issues
- PR #5769[1496] - Slight improvement of small_vector performance
- PR #5766[1497] - Fixing a integral conversion warning
- PR #5765[1498] - Adding a sphinx role allowing to link to a file directly in github
- PR #5763[1499] - add num_cores facility
- PR #5762[1500] - Fix Public API main page
- PR #5761[1501] - Add missing inline to mpi_exception.hpp error_message function
- PR #5760[1502] - Update cdash build url

1480 https://github.com/STEllAR-GROUP/hpx/pull/5792
1481 https://github.com/STEllAR-GROUP/hpx/pull/5791
1482 https://github.com/STEllAR-GROUP/hpx/pull/5790
1483 https://github.com/STEllAR-GROUP/hpx/pull/5787
1484 https://github.com/STEllAR-GROUP/hpx/pull/5786
1485 https://github.com/STEllAR-GROUP/hpx/pull/5785
1486 https://github.com/STEllAR-GROUP/hpx/pull/5783
1487 https://github.com/STEllAR-GROUP/hpx/pull/5781
1488 https://github.com/STEllAR-GROUP/hpx/pull/5780
1489 https://github.com/STEllAR-GROUP/hpx/pull/5778
1490 https://github.com/STEllAR-GROUP/hpx/pull/5777
1491 https://github.com/STEllAR-GROUP/hpx/pull/5775
1492 https://github.com/STEllAR-GROUP/hpx/pull/5773
1493 https://github.com/STEllAR-GROUP/hpx/pull/5772
1494 https://github.com/STEllAR-GROUP/hpx/pull/5771
1495 https://github.com/STEllAR-GROUP/hpx/pull/5770
1496 https://github.com/STEllAR-GROUP/hpx/pull/5769
1497 https://github.com/STEllAR-GROUP/hpx/pull/5766
1498 https://github.com/STEllAR-GROUP/hpx/pull/5765
1499 https://github.com/STEllAR-GROUP/hpx/pull/5763
1500 https://github.com/STEllAR-GROUP/hpx/pull/5762
1501 https://github.com/STEllAR-GROUP/hpx/pull/5761
1502 https://github.com/STEllAR-GROUP/hpx/pull/5760

- PR #5759[1503] - Switch to use generic rostam SLURM partitions
- PR #5758[1504] - Adding support for P2300 completion signatures
- PR #5757[1505] - Fix missing links in Public API
- PR #5756[1506] - Add stop support to when_all
- PR #5755[1507] - Support for data-parallelism for mismatch algorithm
- PR #5754[1508] - Support for data-parallelism for equal algorithm
- PR #5751[1509] - Propagate MPI dependencies to command line handling
- PR #5750[1510] - Make sure required MPI initialization flags are properly applied and supported
- PR #5749[1511] - P2300 stop token
- PR #5748[1512] - Adding environmental query CPOs
- PR #5747[1513] - Renaming set_done to set_stopped (as per P2300)
- PR #5745[1514] - Modernize serialization module
- PR #5743[1515] - Add check for MPICH and set the correct env to support multi-threaded
- PR #5742[1516] - Remove obsolete files related to cpuid, etc.
- PR #5741[1517] - Support for data-parallelism for adjacent find
- PR #5740[1518] - Support for data-parallelism for find algorithms
- PR #5739[1519] - Enable the option to attach a debugger on a segmentation fault (linux)
- PR #5738[1520] - Fixing spell-checking errors
- PR #5737[1521] - Attempt to fix migrate_component issue
- PR #5736[1522] - Set commit status from Jenkins also for special branches
- PR #5734[1523] - Revert #5586
- PR #5732[1524] - Attempt to improve build-id reporting to cdash
- PR #5731[1525] - Randomly delay execution of bash scripts launched by Jenkins

[1503] https://github.com/STEllAR-GROUP/hpx/pull/5759
[1504] https://github.com/STEllAR-GROUP/hpx/pull/5758
[1505] https://github.com/STEllAR-GROUP/hpx/pull/5757
[1506] https://github.com/STEllAR-GROUP/hpx/pull/5756
[1507] https://github.com/STEllAR-GROUP/hpx/pull/5755
[1508] https://github.com/STEllAR-GROUP/hpx/pull/5754
[1509] https://github.com/STEllAR-GROUP/hpx/pull/5751
[1510] https://github.com/STEllAR-GROUP/hpx/pull/5750
[1511] https://github.com/STEllAR-GROUP/hpx/pull/5749
[1512] https://github.com/STEllAR-GROUP/hpx/pull/5748
[1513] https://github.com/STEllAR-GROUP/hpx/pull/5747
[1514] https://github.com/STEllAR-GROUP/hpx/pull/5745
[1515] https://github.com/STEllAR-GROUP/hpx/pull/5743
[1516] https://github.com/STEllAR-GROUP/hpx/pull/5742
[1517] https://github.com/STEllAR-GROUP/hpx/pull/5741
[1518] https://github.com/STEllAR-GROUP/hpx/pull/5740
[1519] https://github.com/STEllAR-GROUP/hpx/pull/5739
[1520] https://github.com/STEllAR-GROUP/hpx/pull/5738
[1521] https://github.com/STEllAR-GROUP/hpx/pull/5737
[1522] https://github.com/STEllAR-GROUP/hpx/pull/5736
[1523] https://github.com/STEllAR-GROUP/hpx/pull/5734
[1524] https://github.com/STEllAR-GROUP/hpx/pull/5732
[1525] https://github.com/STEllAR-GROUP/hpx/pull/5731

- PR #5729[1526] - Workaround for CMake/Ninja generator OOM problem
- PR #5727[1527] - Moving compression plugins to components directory
- PR #5726[1528] - Moving/consolidating parcel coalescing plugin sources
- PR #5725[1529] - Making sure headers for serialization filters are being installed
- PR #5723[1530] - Moving more tests to modules
- PR #5722[1531] - Removing superfluous semicolons
- PR #5720[1532] - Moving parcelports into modules
- PR #5719[1533] - Moving more files to parcelset module
- PR #5718[1534] - build: refactor sphinx config file
- PR #5717[1535] - Creating parcelset modules
- PR #5716[1536] - Avoid duplicate definition error
- PR #5715[1537] - The new LCI parcelport for HPX
- PR #5714[1538] - Refine propagation of **HPX_WITH_**… options
- PR #5713[1539] - Significantly reduce CI jobs run on Piz Daint
- PR #5712[1540] - Updating jenkins configuration for Rostam2.2
- PR #5711[1541] - Refactor manual sections
- PR #5710[1542] - Making task_group serializable
- PR #5709[1543] - Update the MPI cmake setup
- PR #5707[1544] - Better diagnose parcel bootstrap problems
- PR #5704[1545] - Test with hwloc 2.7.0 with GCC 11
- PR #5703[1546] - Fix *counting_iterator* container tests
- PR #5702[1547] - Attempting to fix CircleCI timeouts
- PR #5699[1548] - Update CI to use Boost 1.78.0

[1526] https://github.com/STEllAR-GROUP/hpx/pull/5729
[1527] https://github.com/STEllAR-GROUP/hpx/pull/5727
[1528] https://github.com/STEllAR-GROUP/hpx/pull/5726
[1529] https://github.com/STEllAR-GROUP/hpx/pull/5725
[1530] https://github.com/STEllAR-GROUP/hpx/pull/5723
[1531] https://github.com/STEllAR-GROUP/hpx/pull/5722
[1532] https://github.com/STEllAR-GROUP/hpx/pull/5720
[1533] https://github.com/STEllAR-GROUP/hpx/pull/5719
[1534] https://github.com/STEllAR-GROUP/hpx/pull/5718
[1535] https://github.com/STEllAR-GROUP/hpx/pull/5717
[1536] https://github.com/STEllAR-GROUP/hpx/pull/5716
[1537] https://github.com/STEllAR-GROUP/hpx/pull/5715
[1538] https://github.com/STEllAR-GROUP/hpx/pull/5714
[1539] https://github.com/STEllAR-GROUP/hpx/pull/5713
[1540] https://github.com/STEllAR-GROUP/hpx/pull/5712
[1541] https://github.com/STEllAR-GROUP/hpx/pull/5711
[1542] https://github.com/STEllAR-GROUP/hpx/pull/5710
[1543] https://github.com/STEllAR-GROUP/hpx/pull/5709
[1544] https://github.com/STEllAR-GROUP/hpx/pull/5707
[1545] https://github.com/STEllAR-GROUP/hpx/pull/5704
[1546] https://github.com/STEllAR-GROUP/hpx/pull/5703
[1547] https://github.com/STEllAR-GROUP/hpx/pull/5702
[1548] https://github.com/STEllAR-GROUP/hpx/pull/5699

- PR #5697[1549] - Adding fork_join_executor to foreach_benchmark
- PR #5696[1550] - Modernize when_all and friends (when_any, when_some, when_each)
- PR #5693[1551] - Fix test errors with *_GLIBCXX_DEBUG* defined
- PR #5691[1552] - Rename *annotate_function* to *scoped_annotation*
- PR #5690[1553] - Replace tag_dispatch with tag_invoke in minmax segmented
- PR #5688[1554] - Remove more deprecated macros
- PR #5687[1555] - Add most important CMake options
- PR #5685[1556] - Fix future API
- PR #5684[1557] - Move lock registration to separate module and remove global lock registration
- PR #5683[1558] - Make hpx::wait_all etc. throw exceptions when waited futures hold exceptions and deprecate hpx::lcos::wait_all[_n] in favor of hpx::wait_all[_n]
- PR #5682[1559] - Fix macOS test exceptions
- PR #5681[1560] - docs: add links to hpx recepies
- PR #5680[1561] - Embed base execution policies to datapar execution policies
- PR #5679[1562] - Fix *fork_join_executor* with dynamic schedule
- PR #5678[1563] - Fix compilation of service executors with nvcc
- PR #5677[1564] - Remove compute_cuda module
- PR #5676[1565] - Don't require up-to-date approvals for bors
- PR #5675[1566] - Add default template type parameters for algorithms
- PR #5674[1567] - Allow using *any_sender* in global variables
- PR #5671[1568] - Making sure task_group can be reused
- PR #5670[1569] - Relax constraints on *execution::when_all*
- PR #5669[1570] - Use HPX_WITH_CXX_STANDARD for controlling C++ version
- PR #5667[1571] - Attempt to fix compilation issues with Boost V1.76

[1549] https://github.com/STEllAR-GROUP/hpx/pull/5697
[1550] https://github.com/STEllAR-GROUP/hpx/pull/5696
[1551] https://github.com/STEllAR-GROUP/hpx/pull/5693
[1552] https://github.com/STEllAR-GROUP/hpx/pull/5691
[1553] https://github.com/STEllAR-GROUP/hpx/pull/5690
[1554] https://github.com/STEllAR-GROUP/hpx/pull/5688
[1555] https://github.com/STEllAR-GROUP/hpx/pull/5687
[1556] https://github.com/STEllAR-GROUP/hpx/pull/5685
[1557] https://github.com/STEllAR-GROUP/hpx/pull/5684
[1558] https://github.com/STEllAR-GROUP/hpx/pull/5683
[1559] https://github.com/STEllAR-GROUP/hpx/pull/5682
[1560] https://github.com/STEllAR-GROUP/hpx/pull/5681
[1561] https://github.com/STEllAR-GROUP/hpx/pull/5680
[1562] https://github.com/STEllAR-GROUP/hpx/pull/5679
[1563] https://github.com/STEllAR-GROUP/hpx/pull/5678
[1564] https://github.com/STEllAR-GROUP/hpx/pull/5677
[1565] https://github.com/STEllAR-GROUP/hpx/pull/5676
[1566] https://github.com/STEllAR-GROUP/hpx/pull/5675
[1567] https://github.com/STEllAR-GROUP/hpx/pull/5674
[1568] https://github.com/STEllAR-GROUP/hpx/pull/5671
[1569] https://github.com/STEllAR-GROUP/hpx/pull/5670
[1570] https://github.com/STEllAR-GROUP/hpx/pull/5669
[1571] https://github.com/STEllAR-GROUP/hpx/pull/5667

- PR #5664[1572] - Change logging errors to warnings in schedulers
- PR #5663[1573] - Use dynamic bitsets by default for CPU masks
- PR #5662[1574] - Disambiguate namespace for MSVC
- PR #5660[1575] - Replacing remaining std::forward and std::move with HPX_FORWARD and HPX_MOVE
- PR #5659[1576] - Modernize hpx::future and related facilities
- PR #5658[1577] - Replace HPX_INLINE_CONSTEXPR_VARIABLE with inline constexpr
- PR #5657[1578] - Remove tagged, tagged_pair and tagged_tuple, remove tuple/pair specializations
- PR #5656[1579] - Rename on execution::schedule_from, rename just_on to just_transfer, and add transfer
- PR #5655[1580] - Avoid for module lists to grow indefinitely in cmake cache
- PR #5649[1581] - build: replace usage of Python's reserved words and functions as variable names
- PR #5648[1582] - Modernize action modules and related code
- PR #5646[1583] - Fix ends_with test
- PR #5645[1584] - Add matrix multiplication example
- PR #5644[1585] - Rename execution::transform to execution::then and execution::detach to execution::start_detached
- PR #5643[1586] - Update performance test references
- PR #5642[1587] - Adapting adjacent_difference to work with proxy iterators
- PR #5641[1588] - Factorize perftests scripts
- PR #5640[1589] - Fixed links to sources in Sphinx documentation
- PR #5639[1590] - Fix generate datapar tests for Vc
- PR #5638[1591] - Simd all any none
- PR #5637[1592] - Use bors for merging pull requests
- PR #5636[1593] - Fix leftover std::holds_alternative usage
- PR #5635[1594] - Update container image tag in GitHub actions HIP configuration

1572 https://github.com/STEllAR-GROUP/hpx/pull/5664
1573 https://github.com/STEllAR-GROUP/hpx/pull/5663
1574 https://github.com/STEllAR-GROUP/hpx/pull/5662
1575 https://github.com/STEllAR-GROUP/hpx/pull/5660
1576 https://github.com/STEllAR-GROUP/hpx/pull/5659
1577 https://github.com/STEllAR-GROUP/hpx/pull/5658
1578 https://github.com/STEllAR-GROUP/hpx/pull/5657
1579 https://github.com/STEllAR-GROUP/hpx/pull/5656
1580 https://github.com/STEllAR-GROUP/hpx/pull/5655
1581 https://github.com/STEllAR-GROUP/hpx/pull/5649
1582 https://github.com/STEllAR-GROUP/hpx/pull/5648
1583 https://github.com/STEllAR-GROUP/hpx/pull/5646
1584 https://github.com/STEllAR-GROUP/hpx/pull/5645
1585 https://github.com/STEllAR-GROUP/hpx/pull/5644
1586 https://github.com/STEllAR-GROUP/hpx/pull/5643
1587 https://github.com/STEllAR-GROUP/hpx/pull/5642
1588 https://github.com/STEllAR-GROUP/hpx/pull/5641
1589 https://github.com/STEllAR-GROUP/hpx/pull/5640
1590 https://github.com/STEllAR-GROUP/hpx/pull/5639
1591 https://github.com/STEllAR-GROUP/hpx/pull/5638
1592 https://github.com/STEllAR-GROUP/hpx/pull/5637
1593 https://github.com/STEllAR-GROUP/hpx/pull/5636
1594 https://github.com/STEllAR-GROUP/hpx/pull/5635

- PR #5633[1595] - Moving packaged_task to module futures
- PR #5632[1596] - Tell Asio to use std::aligned_new only if available
- PR #5631[1597] - Adding tag parameter to channel communicator get/set
- PR #5630[1598] - Add partial_sort_copy and adapt partial sort to c++ 20
- PR #5629[1599] - Set HPX_WITH_FETCH_ASIO to OFF as available in the docker image
- PR #5628[1600] - Add Clang 13 CI configuration
- PR #5627[1601] - Replace alternative keyword
- PR #5626[1602] - docs: add support for BibTeX references in Sphinx docs
- PR #5624[1603] - Fix pkgconfig replacements involving CMAKE_INSTALL_PREFIX
- PR #5623[1604] - build: remove unused import from conf.py.in
- PR #5622[1605] - Remove HPX_WITH_VCPKG CMake option
- PR #5621[1606] - Replacing boost::container::small_vector
- PR #5620[1607] - Update Asio tag from 1.18.2 to 1.20.0
- PR #5619[1608] - Fix block_os_threads_1036 test
- PR #5618[1609] - Make sure condition variables are notified under a lock in the thread_pool_scheduler test
- PR #5617[1610] - Use advance_and_get_distance where required
- PR #5616[1611] - Remove separately building segmented algorithms on CircleCI
- PR #5613[1612] - Fix Vc datapar adjacent_difference
- PR #5609[1613] - docs: add anchor links to performance counter tables
- PR #5608[1614] - Fix header test error by adding missing numeric
- PR #5607[1615] - Fix simd adj diff
- PR #5605[1616] - Fix usage of HPX_INVOKE macro
- PR #5604[1617] - Make use of shell-session to allow non-copyable $

[1595] https://github.com/STEllAR-GROUP/hpx/pull/5633
[1596] https://github.com/STEllAR-GROUP/hpx/pull/5632
[1597] https://github.com/STEllAR-GROUP/hpx/pull/5631
[1598] https://github.com/STEllAR-GROUP/hpx/pull/5630
[1599] https://github.com/STEllAR-GROUP/hpx/pull/5629
[1600] https://github.com/STEllAR-GROUP/hpx/pull/5628
[1601] https://github.com/STEllAR-GROUP/hpx/pull/5627
[1602] https://github.com/STEllAR-GROUP/hpx/pull/5626
[1603] https://github.com/STEllAR-GROUP/hpx/pull/5624
[1604] https://github.com/STEllAR-GROUP/hpx/pull/5623
[1605] https://github.com/STEllAR-GROUP/hpx/pull/5622
[1606] https://github.com/STEllAR-GROUP/hpx/pull/5621
[1607] https://github.com/STEllAR-GROUP/hpx/pull/5620
[1608] https://github.com/STEllAR-GROUP/hpx/pull/5619
[1609] https://github.com/STEllAR-GROUP/hpx/pull/5618
[1610] https://github.com/STEllAR-GROUP/hpx/pull/5617
[1611] https://github.com/STEllAR-GROUP/hpx/pull/5616
[1612] https://github.com/STEllAR-GROUP/hpx/pull/5613
[1613] https://github.com/STEllAR-GROUP/hpx/pull/5609
[1614] https://github.com/STEllAR-GROUP/hpx/pull/5608
[1615] https://github.com/STEllAR-GROUP/hpx/pull/5607
[1616] https://github.com/STEllAR-GROUP/hpx/pull/5605
[1617] https://github.com/STEllAR-GROUP/hpx/pull/5604

- PR #5603[1618] - Suppress some MSVC warnings in C++20 mode
- PR #5602[1619] - Test HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE=OFF to one CI configuration
- PR #5601[1620] - Test case for any_sender should use hpx::tuple
- PR #5600[1621] - Rename tag_dispatch back to tag_invoke
- PR #5599[1622] - Change theme, fix Quickstart & Examples
- PR #5596[1623] - Use precompiled headers in tests
- PR #5595[1624] - Drop semicolons for macro calls
- PR #5594[1625] - Adapt datapar generate
- PR #5593[1626] - Update any_sender to use tag_dispatch for execution customizations
- PR #5592[1627] - Add nth_element
- PR #5591[1628] - Remove unnecessary checks for C++17 for tests
- PR #5589[1629] - Add HPX_FORWARD/HPX_MOVE macros
- PR #5588[1630] - Fixing the output formatting for id_types
- PR #5586[1631] - Remove local functionality
- PR #5585[1632] - Delete GitExternal.cmake
- PR #5584[1633] - Serialization of hpx::tuple must use hpx::get
- PR #5583[1634] - fix coroutine_traits allocate calls, add unhandled_exception() implementation.
- PR #5582[1635] - Make more examples work with local runtime
- PR #5581[1636] - Add support for several performance tests in CI
- PR #5580[1637] - Adapt simd adj diff
- PR #5579[1638] - Split absolute paths for generated pkg-config files into -L/-l parts
- PR #5577[1639] - fix unit fill test for datapar with Vc
- PR #5576[1640] - Update forgotten “Full” names

[1618] https://github.com/STEllAR-GROUP/hpx/pull/5603
[1619] https://github.com/STEllAR-GROUP/hpx/pull/5602
[1620] https://github.com/STEllAR-GROUP/hpx/pull/5601
[1621] https://github.com/STEllAR-GROUP/hpx/pull/5600
[1622] https://github.com/STEllAR-GROUP/hpx/pull/5599
[1623] https://github.com/STEllAR-GROUP/hpx/pull/5596
[1624] https://github.com/STEllAR-GROUP/hpx/pull/5595
[1625] https://github.com/STEllAR-GROUP/hpx/pull/5594
[1626] https://github.com/STEllAR-GROUP/hpx/pull/5593
[1627] https://github.com/STEllAR-GROUP/hpx/pull/5592
[1628] https://github.com/STEllAR-GROUP/hpx/pull/5591
[1629] https://github.com/STEllAR-GROUP/hpx/pull/5589
[1630] https://github.com/STEllAR-GROUP/hpx/pull/5588
[1631] https://github.com/STEllAR-GROUP/hpx/pull/5586
[1632] https://github.com/STEllAR-GROUP/hpx/pull/5585
[1633] https://github.com/STEllAR-GROUP/hpx/pull/5584
[1634] https://github.com/STEllAR-GROUP/hpx/pull/5583
[1635] https://github.com/STEllAR-GROUP/hpx/pull/5582
[1636] https://github.com/STEllAR-GROUP/hpx/pull/5581
[1637] https://github.com/STEllAR-GROUP/hpx/pull/5580
[1638] https://github.com/STEllAR-GROUP/hpx/pull/5579
[1639] https://github.com/STEllAR-GROUP/hpx/pull/5577
[1640] https://github.com/STEllAR-GROUP/hpx/pull/5576

- PR #5575[1641] - Change scan partitioner implementation
- PR #5574[1642] - Remove a few deprecated and unused CMake options
- PR #5572[1643] - Remove more guards for the distributed runtime
- PR #5571[1644] - Add workaround for libstc++ in string_util trim
- PR #5569[1645] - Use no_unique_address in sender adaptors
- PR #5568[1646] - Change try catch block to try_catch_exception_ptr
- PR #5567[1647] - Make default_agent::yield actually yield
- PR #5564[1648] - Adjacent
- PR #5562[1649] - More changes to overcome build problems on Windows after recent module rearrangements
- PR #5560[1650] - Update tests and examples
- PR #5559[1651] - Fixing cmake folder names after module restructuring
- PR #5558[1652] - Fixing wrong module dependencies
- PR #5557[1653] - Adding an example for the new channel_communicator API
- PR #5556[1654] - Remove leftover thread pool os executor tests
- PR #5555[1655] - Add option enabling serializing raw pointers
- PR #5554[1656] - Make sure command line aliasing is properly handled
- PR #5552[1657] - Modernizing some of the async facilities
- PR #5551[1658] - Fixing for local executions of actions to properly set task names
- PR #5550[1659] - Update CUDA module in clang-cuda configuration
- PR #5549[1660] - Fixing agent_ref::yield_k to actually call yield_k
- PR #5548[1661] - Making get_action_name() noexcept
- PR #5547[1662] - Fixing communication set
- PR #5546[1663] - Fixing shutdown problems caused by missing ref-counting

[1641] https://github.com/STEllAR-GROUP/hpx/pull/5575
[1642] https://github.com/STEllAR-GROUP/hpx/pull/5574
[1643] https://github.com/STEllAR-GROUP/hpx/pull/5572
[1644] https://github.com/STEllAR-GROUP/hpx/pull/5571
[1645] https://github.com/STEllAR-GROUP/hpx/pull/5569
[1646] https://github.com/STEllAR-GROUP/hpx/pull/5568
[1647] https://github.com/STEllAR-GROUP/hpx/pull/5567
[1648] https://github.com/STEllAR-GROUP/hpx/pull/5564
[1649] https://github.com/STEllAR-GROUP/hpx/pull/5562
[1650] https://github.com/STEllAR-GROUP/hpx/pull/5560
[1651] https://github.com/STEllAR-GROUP/hpx/pull/5559
[1652] https://github.com/STEllAR-GROUP/hpx/pull/5558
[1653] https://github.com/STEllAR-GROUP/hpx/pull/5557
[1654] https://github.com/STEllAR-GROUP/hpx/pull/5556
[1655] https://github.com/STEllAR-GROUP/hpx/pull/5555
[1656] https://github.com/STEllAR-GROUP/hpx/pull/5554
[1657] https://github.com/STEllAR-GROUP/hpx/pull/5552
[1658] https://github.com/STEllAR-GROUP/hpx/pull/5551
[1659] https://github.com/STEllAR-GROUP/hpx/pull/5550
[1660] https://github.com/STEllAR-GROUP/hpx/pull/5549
[1661] https://github.com/STEllAR-GROUP/hpx/pull/5548
[1662] https://github.com/STEllAR-GROUP/hpx/pull/5547
[1663] https://github.com/STEllAR-GROUP/hpx/pull/5546

- PR #5545[1664] - Remove wrong move in thread_pool_scheduler_bulk.hpp
- PR #5543[1665] - Extend launch policy to carry stack size and scheduling hint in addition to priority
- PR #5542[1666] - Simplify execution CPOs
- PR #5540[1667] - Adapt partition, partition_copy and stable_partition to C++ 20
- PR #5539[1668] - Adapt mismatch to support sentinels
- PR #5538[1669] - Document specific sphinx version required for the documentation
- PR #5537[1670] - Test release and debug builds on Piz Daint
- PR #5536[1671] - This fixes referencing stale iterators during the execution of binary mismatch
- PR #5535[1672] - Rename simdpar to par_simd
- PR #5534[1673] - Fix Quick start & Manual Docs
- PR #5533[1674] - Fix *annotate_function* for *std::string*
- PR #5532[1675] - Update two remaining apex links from khuck to UO-OACISS
- PR #5531[1676] - Use contiguous_index_queue in thread_pool_scheduler
- PR #5530[1677] - Eagerly initialize a configurable number of threads on scheduler/thread queue init
- PR #5529[1678] - Update benchmarks and add support for scheduler_executor
- PR #5528[1679] - Add missing properties to executors/schedulers
- PR #5527[1680] - Set local thread/pool number in local/static_queue_scheduler
- PR #5526[1681] - Update Rostam HIP configuration to use 4.3.0
- PR #5525[1682] - Fix Building HPX in Quick start
- PR #5524[1683] - Upload image on cdash
- PR #5523[1684] - Modernize facilities related to hpx::sync
- PR #5522[1685] - Add sender overloads for remaining algorithms
- PR #5521[1686] - Minor changes that improve performance

[1664] https://github.com/STEllAR-GROUP/hpx/pull/5545
[1665] https://github.com/STEllAR-GROUP/hpx/pull/5543
[1666] https://github.com/STEllAR-GROUP/hpx/pull/5542
[1667] https://github.com/STEllAR-GROUP/hpx/pull/5540
[1668] https://github.com/STEllAR-GROUP/hpx/pull/5539
[1669] https://github.com/STEllAR-GROUP/hpx/pull/5538
[1670] https://github.com/STEllAR-GROUP/hpx/pull/5537
[1671] https://github.com/STEllAR-GROUP/hpx/pull/5536
[1672] https://github.com/STEllAR-GROUP/hpx/pull/5535
[1673] https://github.com/STEllAR-GROUP/hpx/pull/5534
[1674] https://github.com/STEllAR-GROUP/hpx/pull/5533
[1675] https://github.com/STEllAR-GROUP/hpx/pull/5532
[1676] https://github.com/STEllAR-GROUP/hpx/pull/5531
[1677] https://github.com/STEllAR-GROUP/hpx/pull/5530
[1678] https://github.com/STEllAR-GROUP/hpx/pull/5529
[1679] https://github.com/STEllAR-GROUP/hpx/pull/5528
[1680] https://github.com/STEllAR-GROUP/hpx/pull/5527
[1681] https://github.com/STEllAR-GROUP/hpx/pull/5526
[1682] https://github.com/STEllAR-GROUP/hpx/pull/5525
[1683] https://github.com/STEllAR-GROUP/hpx/pull/5524
[1684] https://github.com/STEllAR-GROUP/hpx/pull/5523
[1685] https://github.com/STEllAR-GROUP/hpx/pull/5522
[1686] https://github.com/STEllAR-GROUP/hpx/pull/5521

- PR #5520[1687] - Update reference as perftests failing regularly
- PR #5519[1688] - Add transform_mpi sender adapter
- PR #5518[1689] - Add sender overloads to rotate/rotate_copy
- PR #5517[1690] - Fix coroutine integration
- PR #5515[1691] - Avoid deadlock in ignore_while_locked_1485 test
- PR #5514[1692] - Add split sender adapter
- PR #5512[1693] - Update Rostam HIP configuration
- PR #5511[1694] - Fix Asio target name for precompiled headers
- PR #5510[1695] - Add any_sender and unique_any_sender
- PR #5509[1696] - Test with Boost 1.77 on gcc/clang-newest configurations
- PR #5508[1697] - Minor release changes from 1.7.1
- PR #5507[1698] - Add missing commits from scheduler_executor PR
- PR #5506[1699] - Fix condition for checking if we should use our own variant
- PR #5501[1700] - Attempt to fix thread_pool_scheduler test
- PR #5493[1701] - Update Jenkins GitHub token to use StellarBot GitHub account
- PR #5490[1702] - Fix clang-format error on master
- PR #5487[1703] - Add get_completion_scheduler CPO and customize bulk for thread_pool_scheduler
- PR #5484[1704] - Add missing header to jacobi_component/server/solver.hpp
- PR #5481[1705] - Changing the APEX repository to the new location
- PR #5479[1706] - Fix version check for CUDA noexcept/result_of bug
- PR #5477[1707] - Require cxx17 minor comments
- PR #5476[1708] - Fix cmake format error
- PR #5475[1709] - Require CMake 3.18 as it is already a requirement for CUDA

[1687] https://github.com/STEllAR-GROUP/hpx/pull/5520
[1688] https://github.com/STEllAR-GROUP/hpx/pull/5519
[1689] https://github.com/STEllAR-GROUP/hpx/pull/5518
[1690] https://github.com/STEllAR-GROUP/hpx/pull/5517
[1691] https://github.com/STEllAR-GROUP/hpx/pull/5515
[1692] https://github.com/STEllAR-GROUP/hpx/pull/5514
[1693] https://github.com/STEllAR-GROUP/hpx/pull/5512
[1694] https://github.com/STEllAR-GROUP/hpx/pull/5511
[1695] https://github.com/STEllAR-GROUP/hpx/pull/5510
[1696] https://github.com/STEllAR-GROUP/hpx/pull/5509
[1697] https://github.com/STEllAR-GROUP/hpx/pull/5508
[1698] https://github.com/STEllAR-GROUP/hpx/pull/5507
[1699] https://github.com/STEllAR-GROUP/hpx/pull/5506
[1700] https://github.com/STEllAR-GROUP/hpx/pull/5501
[1701] https://github.com/STEllAR-GROUP/hpx/pull/5493
[1702] https://github.com/STEllAR-GROUP/hpx/pull/5490
[1703] https://github.com/STEllAR-GROUP/hpx/pull/5487
[1704] https://github.com/STEllAR-GROUP/hpx/pull/5484
[1705] https://github.com/STEllAR-GROUP/hpx/pull/5481
[1706] https://github.com/STEllAR-GROUP/hpx/pull/5479
[1707] https://github.com/STEllAR-GROUP/hpx/pull/5477
[1708] https://github.com/STEllAR-GROUP/hpx/pull/5476
[1709] https://github.com/STEllAR-GROUP/hpx/pull/5475

- PR #5474[1710] - Make the cuda parameters of try_compile optional
- PR #5473[1711] - Update cuda arch and change cuda version
- PR #5471[1712] - Add corrected citation.cff
- PR #5470[1713] - Adapt stable_sort to C++ 20
- PR #5468[1714] - Experimentation to make the perftest report public
- PR #5466[1715] - Add shift_left and shift_right algorithms
- PR #5465[1716] - Adapt datapar fill
- PR #5464[1717] - Moving tag_dispatch to separate module
- PR #5461[1718] - Rename HPX_WITH_CUDA_COMPUTE with HPX_WITH_COMPUTE_CUDA
- PR #5460[1719] - Adapt sort to C++ 20
- PR #5459[1720] - Adapt rotate/rotate_copy to C++20
- PR #5458[1721] - Adapt unique and unique_copy to C++ 20
- PR #5455[1722] - Remove and clean up fallback sender implementations
- PR #5454[1723] - Make performance plot show even if similar performance
- PR #5453[1724] - Post 1.7.0 version bump
- PR #5452[1725] - Fix find_end parallel overload
- PR #5450[1726] - Change the print-bind output to be more precise
- PR #5449[1727] - Adapt swap_ranges to C++ 20
- PR #5446[1728] - Use more verbose names in sender algorithms
- PR #5443[1729] - Properly support ASAN with MSVC
- PR #5441[1730] - Adding reference counting to thread_data
- PR #5429[1731] - Scheduler executor
- PR #5428[1732] - Adapt datapar copy

[1710] https://github.com/STEllAR-GROUP/hpx/pull/5474
[1711] https://github.com/STEllAR-GROUP/hpx/pull/5473
[1712] https://github.com/STEllAR-GROUP/hpx/pull/5471
[1713] https://github.com/STEllAR-GROUP/hpx/pull/5470
[1714] https://github.com/STEllAR-GROUP/hpx/pull/5468
[1715] https://github.com/STEllAR-GROUP/hpx/pull/5466
[1716] https://github.com/STEllAR-GROUP/hpx/pull/5465
[1717] https://github.com/STEllAR-GROUP/hpx/pull/5464
[1718] https://github.com/STEllAR-GROUP/hpx/pull/5461
[1719] https://github.com/STEllAR-GROUP/hpx/pull/5460
[1720] https://github.com/STEllAR-GROUP/hpx/pull/5459
[1721] https://github.com/STEllAR-GROUP/hpx/pull/5458
[1722] https://github.com/STEllAR-GROUP/hpx/pull/5455
[1723] https://github.com/STEllAR-GROUP/hpx/pull/5454
[1724] https://github.com/STEllAR-GROUP/hpx/pull/5453
[1725] https://github.com/STEllAR-GROUP/hpx/pull/5452
[1726] https://github.com/STEllAR-GROUP/hpx/pull/5450
[1727] https://github.com/STEllAR-GROUP/hpx/pull/5449
[1728] https://github.com/STEllAR-GROUP/hpx/pull/5446
[1729] https://github.com/STEllAR-GROUP/hpx/pull/5443
[1730] https://github.com/STEllAR-GROUP/hpx/pull/5441
[1731] https://github.com/STEllAR-GROUP/hpx/pull/5429
[1732] https://github.com/STEllAR-GROUP/hpx/pull/5428

- PR #5421[1733] - Update CI base image to use clang-format 11
- PR #5410[1734] - Add ranges starts_with and ends_with algorithms
- PR #5383[1735] - Tentatively remove runtime_registration_wrapper from cuda futures
- PR #5377[1736] - Fewer Asio includes and more precompiled headers
- PR #5329[1737] - Sender overloads for parallel algorithms
- PR #5313[1738] - Rearrange modules between libraries
- PR #5283[1739] - Require minimum C++17 and change CUDA handling
- PR #5241[1740] - Adapt min_element, max_element and minmax_element to C++20

HPX V1.7.1 (Aug 12, 2021)

This is a bugfix release with a few minor fixes.

General changes

- Added a CMake option to assume that all types are bitwise serializable by default: `HPX_SERIALIZATION_WITH_ALL_TYPES_ARE_BITWISE_SERIALIZABLE`. The default value `OFF` corresponds to the old behaviour.
- Added a version check for Asio. The minimum Asio version supported by *HPX* is 1.12.0.
- Fixed a bug affecting usage of actions, where the internals of *HPX* relied on function addresses being unique. This was fixed by relying on variable addresses being unique instead.
- Made `hpx::util::bind` more strict in checking the validity of placeholders.
- Small performance improvement to spinlocks.
- Adapted the following parallel algorithms to C++20: `inclusive_scan`, `exclusive_scan`, `transform_inclusive_scan`, `transform_exclusive_scan`.

Breaking changes

- The experimental `hpx::execution::simdpar` execution policy (introduced in 1.7.0) was renamed to `hpx::execution::par_simd` for consistency with the other parallel policies.

[1733] https://github.com/STEllAR-GROUP/hpx/pull/5421
[1734] https://github.com/STEllAR-GROUP/hpx/pull/5410
[1735] https://github.com/STEllAR-GROUP/hpx/pull/5383
[1736] https://github.com/STEllAR-GROUP/hpx/pull/5377
[1737] https://github.com/STEllAR-GROUP/hpx/pull/5329
[1738] https://github.com/STEllAR-GROUP/hpx/pull/5313
[1739] https://github.com/STEllAR-GROUP/hpx/pull/5283
[1740] https://github.com/STEllAR-GROUP/hpx/pull/5241

Closed issues

- Issue #5494[1741] - Rename *simdpar* execution policy to *par_simd*
- Issue #5488[1742] - *hpx::util::bind* doesn't bounds-check placeholders
- Issue #5486[1743] - Possible V1.7.1 release

Closed pull requests

- PR #5500[1744] - Minor bug fix in transform exclusive and inclusive scan tests
- PR #5499[1745] - Rename simdpar to par_simd
- PR #5489[1746] - Adding bound-checking for bind placeholders
- PR #5485[1747] - Add Asio version check
- PR #5482[1748] - Change extra archive data to rely on uniqueness of a variable address, not a function address
- PR #5448[1749] - More fixes to enable for all types to be assumed to be bitwise copyable
- PR #5445[1750] - Improve performance of Spinlocks
- PR #5444[1751] - Adapt transform_inclusive_scan to C++ 20
- PR #5440[1752] - Adapt transform_exclusive_scan to C++ 20
- PR #5439[1753] - Adapt inclusive_scan to C++ 20
- PR #5436[1754] - Adapt exclusive_scan to C++20

HPX V1.7.0 (Jul 14, 2021)

This release is again focused on C++20 conformance of algorithms. Additionally, many new experimental sender-based algorithms have been added based on the latest proposals.

[1741] https://github.com/STEllAR-GROUP/hpx/issues/5494
[1742] https://github.com/STEllAR-GROUP/hpx/issues/5488
[1743] https://github.com/STEllAR-GROUP/hpx/issues/5486
[1744] https://github.com/STEllAR-GROUP/hpx/pull/5500
[1745] https://github.com/STEllAR-GROUP/hpx/pull/5499
[1746] https://github.com/STEllAR-GROUP/hpx/pull/5489
[1747] https://github.com/STEllAR-GROUP/hpx/pull/5485
[1748] https://github.com/STEllAR-GROUP/hpx/pull/5482
[1749] https://github.com/STEllAR-GROUP/hpx/pull/5448
[1750] https://github.com/STEllAR-GROUP/hpx/pull/5445
[1751] https://github.com/STEllAR-GROUP/hpx/pull/5444
[1752] https://github.com/STEllAR-GROUP/hpx/pull/5440
[1753] https://github.com/STEllAR-GROUP/hpx/pull/5439
[1754] https://github.com/STEllAR-GROUP/hpx/pull/5436

General changes

- The following algorithms have been adapted to be C++20 conformant:
  - `remove`,
  - `remove_if`,
  - `remove_copy`,
  - `remove_copy_if`,
  - `replace`,
  - `replace_if`,
  - `reverse`, and
  - `lexicographical_compare`.
- When the compiler and standard library support the standard execution policies `std::execution::seq`, `std::execution::par`, and `std::execution::par_unseq` they can now be used in all *HPX* parallel algorithms with equivalent behaviour to the non-task policies `hpx::execution::seq`, `hpx::execution::par`, and `hpx::execution::par_unseq`.
- Vc support has been fixed, after being broken in 1.6.0. In addition, *HPX* now experimentally supports GCC's SIMD implementation, when available. The implementation can be used through the `hpx::execution::simd` and `hpx::execution::simdpar` execution policies.
- The customization points `sync_execute`, `async_execute`, `then_execute`, `post`, `bulk_sync_execute`, `bulk_async_execute`, and `bulk_then_execute` are now implemented using `tag_dispatch` (previously `tag_invoke`). Executors can still be implemented by providing the aforementioned functions as member functions of an executor.
- New functionality, enhancements, and fixes based on P0443r14 (executors proposal) and P1897 (sender-based algorithms) have been added to the `hpx::execution::experimental` namespace. These can be accessed through the `hpx/execution.hpp` and `hpx/local/execution.hpp` headers. In particular, the following sender-based algorithms have been added:
  - `detach`,
  - `ensure_started`,
  - `just`,
  - `just_on`,
  - `let_error`,
  - `let_value`,
  - `on`,
  - `transform`, and
  - `when_all`.

  Additionally, futures now implement the sender concept. `make_future` can be used to turn a sender into a future. All functionality is experimental and can change without notice.
- All `hpx::init` and `hpx::start` overloads now take `std::functions` instead of `hpx::util::function_nonser`. No changes should be required in user code to accommodate this change.

- `hpx::util::unwrapping` and other related unwrapping functionality has been moved up into the `hpx` namespace. Names in `hpx::util` are still usable with a deprecation warning. This functionality can now be accessed through the `hpx/unwrap.hpp` and `hpx/local/unwrap.hpp` headers.
- The default tag for APEX has been update from 2.3.1 to 2.4.0. In particular, this fixes a bug which could lead to hangs in distributed runs.
- The dependency on Boost.Asio has been replaced with the standalone Asio available at https://github.com/chriskohlhoff/asio. By default, a system-installed Asio will be used. `ASIO_ROOT` can be given as a hint to tell CMake where to find Asio. Alternatively, Asio can be fetched automatically using CMake's fetchcontent by setting `HPX_WITH_FETCH_ASIO=ON`. In general, dependencies on Boost have again been reduced.
- Modularization of the library has continued. In this release almost all functionality has been moved into modules. These changes do not generally affect user code. Warnings are still issued for headers that have moved.
- hipBLAS is now optional when compiling with `hipcc`. A warning instead of an error will be printed if hipBLAS is not found during configuration.
- Previously `HPX_COMPUTE_HOST_CODE` was defined in host code only if HPX was configured with CUDA or HIP. In this release `HPX_COMPUTE_HOST_CODE` is always defined in host code.
- An experimental `HPX_WITH_PRECOMPILED_HEADERS` CMake option has been added to use precompiled headers when building *HPX*. This option should not be used on Windows.
- Numerous bug fixes.

Breaking changes

- The minimum required CMake version is now 3.17.
- The minimum required Boost version is now 1.71.0.
- The customization mechanism used to implement and extend sender functionality and algorithms has been renamed from `tag_invoke` to `tag_dispatch`. All customization of sender functionality should be done by overloading `tag_dispatch`.
- The following compatibility options have been removed, along with their compatibility implementations: - `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY` - `HPX_WITH_ACTION_BASE_COMPATIBILITY` - `HPX_WITH_EMBEDDED_THREAD_POOLS_COMPATIBILITY` - `HPX_WITH_POOL_EXECUTOR_COMPATIBILITY`. - `HPX_WITH_PROMISE_ALIAS_COMPATIBILITY` - `HPX_WITH_REGISTER_THREAD_COMPATIBILITY` - `HPX_WITH_REGISTER_THREAD_OVERLOADS_COMPATIBILITY` - `HPX_WITH_THREAD_AWARE_TIMER_COMPATIBILITY` - `HPX_WITH_THREAD_EXECUTORS_COMPATIBILITY` - `HPX_WITH_THREAD_POOL_OS_EXECUTOR_COMPATIBILITY`
- The `HPX_WITH_THREAD_SCHEDULERS` CMake option has been removed. All schedulers are now enabled when possible.
- `HPX_WITH_INIT_START_OVERLOADS_COMPATIBILITY` has been turned off by default.

Closed issues

- Issue #5423[1755] - Fix lvalue-ref qualified connect for `when_all-sender`
- Issue #5412[1756] - Link error
- Issue #5397[1757] - Performance regression in thread annotations
- Issue #5395[1758] - HPX 1.7.0-rc1 fails to build icw APEX + OTF2
- Issue #5385[1759] - HPX 1.7 crashes on Piz Daint > 64 nodes
- Issue #5380[1760] - CMake should search for asio package installed on the system
- Issue #5378[1761] - HPX 1.7.0 stopped building on Fedora
- Issue #5369[1762] - HPX 1.6 and master hangs on Summit for > 64 nodes
- Issue #5358[1763] - HPX init fails for single-core environments
- Issue #5345[1764] - Rename P2220 property CPOs?
- Issue #5333[1765] - HPX does not compile on the new Mac OSX using the M1 chip
- Issue #5317[1766] - Consider making hipblas optional
- Issue #5306[1767] - asio fails to build with CUDA 10.0
- Issue #5294[1768] - `execution::on` should be based on `execution::schedule`
- Issue #5275[1769] - HPX V1.6.0 fails on Fedora release
- Issue #5270[1770] - HPX-1.6.0 fails to build on Windows 10
- Issue #5257[1771] - Allow triggering the output of OS thread affinity from configuration settings
- Issue #5246[1772] - HPX fails to build on ppc64le
- Issue #5232[1773] - Annotation using `hpx::util::annotated_function` not working
- Issue #5222[1774] - Build and link errors with ittnotify enabled
- Issue #5204[1775] - Move algorithms to tag_fallback_dispatch
- Issue #5163[1776] - Remove module-specific compatibility and deprecation options

[1755] https://github.com/STEllAR-GROUP/hpx/issues/5423
[1756] https://github.com/STEllAR-GROUP/hpx/issues/5412
[1757] https://github.com/STEllAR-GROUP/hpx/issues/5397
[1758] https://github.com/STEllAR-GROUP/hpx/issues/5395
[1759] https://github.com/STEllAR-GROUP/hpx/issues/5385
[1760] https://github.com/STEllAR-GROUP/hpx/issues/5380
[1761] https://github.com/STEllAR-GROUP/hpx/issues/5378
[1762] https://github.com/STEllAR-GROUP/hpx/issues/5369
[1763] https://github.com/STEllAR-GROUP/hpx/issues/5358
[1764] https://github.com/STEllAR-GROUP/hpx/issues/5345
[1765] https://github.com/STEllAR-GROUP/hpx/issues/5333
[1766] https://github.com/STEllAR-GROUP/hpx/issues/5317
[1767] https://github.com/STEllAR-GROUP/hpx/issues/5306
[1768] https://github.com/STEllAR-GROUP/hpx/issues/5294
[1769] https://github.com/STEllAR-GROUP/hpx/issues/5275
[1770] https://github.com/STEllAR-GROUP/hpx/issues/5270
[1771] https://github.com/STEllAR-GROUP/hpx/issues/5257
[1772] https://github.com/STEllAR-GROUP/hpx/issues/5246
[1773] https://github.com/STEllAR-GROUP/hpx/issues/5232
[1774] https://github.com/STEllAR-GROUP/hpx/issues/5222
[1775] https://github.com/STEllAR-GROUP/hpx/issues/5204
[1776] https://github.com/STEllAR-GROUP/hpx/issues/5163

- Issue #5161[1777] - Bump required CMake version to 3.17
- Issue #5143[1778] - Searching for HPX-Application to generate work on multiple Nodes

Closed pull requests

- PR #5438[1779] - Delete datapar/foreach_tests.hpp
- PR #5437[1780] - Add back explicit -pthread flags when available
- PR #5435[1781] - This adds support for systems that assume all types are bitwise serializable by default
- PR #5434[1782] - Update CUDA polling logging to be more verbose
- PR #5433[1783] - Fix `when_all_sender` connect for references
- PR #5432[1784] - Add deprecation warnings for v1.8
- PR #5431[1785] - Rename the new P0443/P2300 executor to `thread_pool_scheduler`
- PR #5430[1786] - Revert "Adding the missing defined for `HPX_HAVE_DEPRECATION_WARNINGS`"
- PR #5427[1787] - Removing unneeded typedef
- PR #5426[1788] - Adding more concept checks for sender/receiver algorithms
- PR #5425[1789] - Adding the missing defined for `HPX_HAVE_DEPRECATION_WARNINGS`
- PR #5424[1790] - Disable Vc in final docker image created in CI
- PR #5422[1791] - Adding `execution::experimental::bulk` algorithm
- PR #5420[1792] - Update logic to find threading library
- PR #5418[1793] - Reduce max size and number of files in ccache cache
- PR #5417[1794] - Final release notes for 1.7.0
- PR #5416[1795] - Adapt `uninitialized_value_construct` and `uninitialized_value_construct_n` to C++ 20
- PR #5415[1796] - Adapt `uninitialized_default_construct` and `uninitialized_default_construct_n` to C++ 20
- PR #5414[1797] - Improve integration of futures and senders

[1777] https://github.com/STEllAR-GROUP/hpx/issues/5161
[1778] https://github.com/STEllAR-GROUP/hpx/issues/5143
[1779] https://github.com/STEllAR-GROUP/hpx/pull/5438
[1780] https://github.com/STEllAR-GROUP/hpx/pull/5437
[1781] https://github.com/STEllAR-GROUP/hpx/pull/5435
[1782] https://github.com/STEllAR-GROUP/hpx/pull/5434
[1783] https://github.com/STEllAR-GROUP/hpx/pull/5433
[1784] https://github.com/STEllAR-GROUP/hpx/pull/5432
[1785] https://github.com/STEllAR-GROUP/hpx/pull/5431
[1786] https://github.com/STEllAR-GROUP/hpx/pull/5430
[1787] https://github.com/STEllAR-GROUP/hpx/pull/5427
[1788] https://github.com/STEllAR-GROUP/hpx/pull/5426
[1789] https://github.com/STEllAR-GROUP/hpx/pull/5425
[1790] https://github.com/STEllAR-GROUP/hpx/pull/5424
[1791] https://github.com/STEllAR-GROUP/hpx/pull/5422
[1792] https://github.com/STEllAR-GROUP/hpx/pull/5420
[1793] https://github.com/STEllAR-GROUP/hpx/pull/5418
[1794] https://github.com/STEllAR-GROUP/hpx/pull/5417
[1795] https://github.com/STEllAR-GROUP/hpx/pull/5416
[1796] https://github.com/STEllAR-GROUP/hpx/pull/5415
[1797] https://github.com/STEllAR-GROUP/hpx/pull/5414

- PR #5413[1798] - Fixing sender/receiver code base to compile with MSVC
- PR #5407[1799] - Handle exceptions thrown during initialization of parcel handler
- PR #5406[1800] - Simplify dispatching to annotation handlers
- PR #5405[1801] - Fetch Asio automatically in perftests CI
- PR #5403[1802] - Create generic executor that adds annotations to any other executor
- PR #5402[1803] - Adapt `uninitialized_fill` and `uninitialized_fill_n` to C++ 20
- PR #5401[1804] - Modernize a variety of facilities related to parallel algorithms
- PR #5400[1805] - Fix sliding semaphore test
- PR #5399[1806] - Rename leftover `tag_fallback_invoke` to `tag_fallback_dispatch`
- PR #5398[1807] - Improve logging in AGAS symbol namespace
- PR #5396[1808] - Introduce compatibility layer for collective operations
- PR #5394[1809] - Enable OTF2 in APEX CI configuration
- PR #5393[1810] - Update APEX tag
- PR #5392[1811] - Fixing wrong usage of `std::forward`
- PR #5391[1812] - Fix forwarding in transform_receiver constructor
- PR #5390[1813] - Make sure shared priority scheduler steals tasks on the current NUMA domain when (core) stealing is enabled
- PR #5389[1814] - Adapt `uninitialized_move` and `uninitialized_move_n` to C++ 20
- PR #5388[1815] - Fixing `gather_there` for used with lvalue reference argument
- PR #5387[1816] - Extend thread state logging and change default stealing parameters
- PR #5386[1817] - Attempt to fix the startup hang with nodes > 32
- PR #5384[1818] - Remove HPX 1.5.0 deprecations
- PR #5382[1819] - Prefer installed Asio before considering FetchContent
- PR #5379[1820] - Allow using pre-downloaded (not installed) versions of Asio and/or Apex

[1798] https://github.com/STEllAR-GROUP/hpx/pull/5413
[1799] https://github.com/STEllAR-GROUP/hpx/pull/5407
[1800] https://github.com/STEllAR-GROUP/hpx/pull/5406
[1801] https://github.com/STEllAR-GROUP/hpx/pull/5405
[1802] https://github.com/STEllAR-GROUP/hpx/pull/5403
[1803] https://github.com/STEllAR-GROUP/hpx/pull/5402
[1804] https://github.com/STEllAR-GROUP/hpx/pull/5401
[1805] https://github.com/STEllAR-GROUP/hpx/pull/5400
[1806] https://github.com/STEllAR-GROUP/hpx/pull/5399
[1807] https://github.com/STEllAR-GROUP/hpx/pull/5398
[1808] https://github.com/STEllAR-GROUP/hpx/pull/5396
[1809] https://github.com/STEllAR-GROUP/hpx/pull/5394
[1810] https://github.com/STEllAR-GROUP/hpx/pull/5393
[1811] https://github.com/STEllAR-GROUP/hpx/pull/5392
[1812] https://github.com/STEllAR-GROUP/hpx/pull/5391
[1813] https://github.com/STEllAR-GROUP/hpx/pull/5390
[1814] https://github.com/STEllAR-GROUP/hpx/pull/5389
[1815] https://github.com/STEllAR-GROUP/hpx/pull/5388
[1816] https://github.com/STEllAR-GROUP/hpx/pull/5387
[1817] https://github.com/STEllAR-GROUP/hpx/pull/5386
[1818] https://github.com/STEllAR-GROUP/hpx/pull/5384
[1819] https://github.com/STEllAR-GROUP/hpx/pull/5382
[1820] https://github.com/STEllAR-GROUP/hpx/pull/5379

- PR #5376[1821] - Remove unnecessary explicit listing of library modules.rst files in CMakeLists.txt
- PR #5375[1822] - Slight performance improvement for `hpx::copy` and `hpx::move` et.al.
- PR #5374[1823] - Remove unnecessary moves from future sender implementations
- PR #5373[1824] - More changes to clang-cuda Jenkins configuration
- PR #5372[1825] - Slight improvements to `min/max/minmax_element` algorithms
- PR #5371[1826] - Adapt `uninitialized_copy` and `uninitialized_copy_n` to C++ 20
- PR #5370[1827] - Decay types in `just_sender` `value_types` to match stored types
- PR #5367[1828] - Disable pkgconfig by default again on macOS
- PR #5365[1829] - Use ccache for Jenkins builds on Piz Daint
- PR #5363[1830] - Update cudatoolkit module name in clang-cuda Jenkins configuration
- PR #5362[1831] - Adding `channel_communicator`
- PR #5361[1832] - Fix compilation with MPI enabled
- PR #5360[1833] - Update APEX and asio tags
- PR #5359[1834] - Fix check for pu-step in single-core case
- PR #5357[1835] - Making sure collective operations can be reused by preallocating communicator
- PR #5356[1836] - Update API documentation
- PR #5355[1837] - Make the `sequenced_executor` `processing_units_count` member function const
- PR #5354[1838] - Making sure `default_stack_size` is defined whenever declared
- PR #5353[1839] - Add CUDA timestamp support to HPX Hardware Clock
- PR #5352[1840] - Adding missing includes
- PR #5351[1841] - Adding `enable_logging/disable_logging` API functions
- PR #5350[1842] - Adapt lexicographical_compare to C++20
- PR #5349[1843] - Update minimum boost version needed on the docs

[1821] https://github.com/STEllAR-GROUP/hpx/pull/5376
[1822] https://github.com/STEllAR-GROUP/hpx/pull/5375
[1823] https://github.com/STEllAR-GROUP/hpx/pull/5374
[1824] https://github.com/STEllAR-GROUP/hpx/pull/5373
[1825] https://github.com/STEllAR-GROUP/hpx/pull/5372
[1826] https://github.com/STEllAR-GROUP/hpx/pull/5371
[1827] https://github.com/STEllAR-GROUP/hpx/pull/5370
[1828] https://github.com/STEllAR-GROUP/hpx/pull/5367
[1829] https://github.com/STEllAR-GROUP/hpx/pull/5365
[1830] https://github.com/STEllAR-GROUP/hpx/pull/5363
[1831] https://github.com/STEllAR-GROUP/hpx/pull/5362
[1832] https://github.com/STEllAR-GROUP/hpx/pull/5361
[1833] https://github.com/STEllAR-GROUP/hpx/pull/5360
[1834] https://github.com/STEllAR-GROUP/hpx/pull/5359
[1835] https://github.com/STEllAR-GROUP/hpx/pull/5357
[1836] https://github.com/STEllAR-GROUP/hpx/pull/5356
[1837] https://github.com/STEllAR-GROUP/hpx/pull/5355
[1838] https://github.com/STEllAR-GROUP/hpx/pull/5354
[1839] https://github.com/STEllAR-GROUP/hpx/pull/5353
[1840] https://github.com/STEllAR-GROUP/hpx/pull/5352
[1841] https://github.com/STEllAR-GROUP/hpx/pull/5351
[1842] https://github.com/STEllAR-GROUP/hpx/pull/5350
[1843] https://github.com/STEllAR-GROUP/hpx/pull/5349

- PR #5348[1844] - Rename `tag_invoke` and related facilities to `tag_dispatch`
- PR #5347[1845] - Remove `make_` prefix for executor properties
- PR #5346[1846] - Remove and disable compatibility options for 1.7.0
- PR #5343[1847] - Fix timed_executor static cast conversion
- PR #5342[1848] - Refactor CUDA event polling
- PR #5341[1849] - Adding `make_with_annotation` and `get_annotation` properties
- PR #5339[1850] - Making sure `hpx::util::hardware::timestamp()` is always defined
- PR #5338[1851] - Fixing `timed_executor` specializations of customization points
- PR #5335[1852] - Make `partial_algorithm` work with any number of arguments
- PR #5334[1853] - Follow up `iter_sent` include on #5225
- PR #5332[1854] - Simplify `tag_invoke` and friends
- PR #5331[1855] - More work on cleaning up executor CPOs
- PR #5330[1856] - Add option to disable pkgconfig generation
- PR #5328[1857] - Adapt data parallel support using std-simd
- PR #5327[1858] - Fix missing `ifdef HPX_SMT_PAUSE`
- PR #5326[1859] - Adding `resize()` to `serialize_buffer` allowing to shrink its size
- PR #5324[1860] - Add get member functions to `async_rw_mutex` proxy objects for explicitly getting the wrapped value
- PR #5323[1861] - Add `keep_future` algorithm
- PR #5322[1862] - Replace executor customization point implementations with `tag_invoke`
- PR #5321[1863] - Seperate segmented algorithms for reduce
- PR #5320[1864] - Fix `is_sender` trait and other small fixes to p0443 traits
- PR #5319[1865] - gcc 11.1 c++20 build fixes
- PR #5318[1866] - Make hipblas dependency optional as not always available

1844 https://github.com/STEllAR-GROUP/hpx/pull/5348
1845 https://github.com/STEllAR-GROUP/hpx/pull/5347
1846 https://github.com/STEllAR-GROUP/hpx/pull/5346
1847 https://github.com/STEllAR-GROUP/hpx/pull/5343
1848 https://github.com/STEllAR-GROUP/hpx/pull/5342
1849 https://github.com/STEllAR-GROUP/hpx/pull/5341
1850 https://github.com/STEllAR-GROUP/hpx/pull/5339
1851 https://github.com/STEllAR-GROUP/hpx/pull/5338
1852 https://github.com/STEllAR-GROUP/hpx/pull/5335
1853 https://github.com/STEllAR-GROUP/hpx/pull/5334
1854 https://github.com/STEllAR-GROUP/hpx/pull/5332
1855 https://github.com/STEllAR-GROUP/hpx/pull/5331
1856 https://github.com/STEllAR-GROUP/hpx/pull/5330
1857 https://github.com/STEllAR-GROUP/hpx/pull/5328
1858 https://github.com/STEllAR-GROUP/hpx/pull/5327
1859 https://github.com/STEllAR-GROUP/hpx/pull/5326
1860 https://github.com/STEllAR-GROUP/hpx/pull/5324
1861 https://github.com/STEllAR-GROUP/hpx/pull/5323
1862 https://github.com/STEllAR-GROUP/hpx/pull/5322
1863 https://github.com/STEllAR-GROUP/hpx/pull/5321
1864 https://github.com/STEllAR-GROUP/hpx/pull/5320
1865 https://github.com/STEllAR-GROUP/hpx/pull/5319
1866 https://github.com/STEllAR-GROUP/hpx/pull/5318

- PR #5316[1867] - Attempt to fix checking for libatomic
- PR #5315[1868] - Add explicit keyword to fixture constructor
- PR #5314[1869] - Fix a race condition in async mpi affecting limiting executor
- PR #5312[1870] - Use local runtime and local headers in local-only modules and tests
- PR #5311[1871] - Add GCC 11 builder to jenkins
- PR #5310[1872] - Adding `hpx::execution::experimental::task_group`
- PR #5309[1873] - Seperate datapar
- PR #5308[1874] - Seperate segmented algorithms for `find`, `find_if`, `find_if_not`
- PR #5307[1875] - Seperate segmented algorithms for `fill` and `generate`
- PR #5304[1876] - Fix compilation of sender CPOs with nvcc
- PR #5300[1877] - Remove `PRIVATE` flag that was propagated into the `LANGUAGES`
- PR #5298[1878] - Seperate datapar
- PR #5297[1879] - Specify exact cmake and ninja versions when loading them in jenkins jobs
- PR #5295[1880] - Update clang-newest configuration to use clang 12 and Boost 1.76.0
- PR #5293[1881] - Fix Clang 11 cuda_future test bug
- PR #5292[1882] - Add `async_rw_mutex` based on senders
- PR #5291[1883] - “Fix” termination detection
- PR #5290[1884] - Fixed source file line statements in examples documentation
- PR #5289[1885] - Allow splitting of futures holding `std::tuple`
- PR #5288[1886] - Move algorithms to `tag_fallback_invoke`
- PR #5287[1887] - Move algorithms to `tag_fallback_invoke`
- PR #5285[1888] - Fix clang-format failure on master
- PR #5284[1889] - Replacing `util::function_nonser` on std::function in `hpx_init`

[1867] https://github.com/STEllAR-GROUP/hpx/pull/5316
[1868] https://github.com/STEllAR-GROUP/hpx/pull/5315
[1869] https://github.com/STEllAR-GROUP/hpx/pull/5314
[1870] https://github.com/STEllAR-GROUP/hpx/pull/5312
[1871] https://github.com/STEllAR-GROUP/hpx/pull/5311
[1872] https://github.com/STEllAR-GROUP/hpx/pull/5310
[1873] https://github.com/STEllAR-GROUP/hpx/pull/5309
[1874] https://github.com/STEllAR-GROUP/hpx/pull/5308
[1875] https://github.com/STEllAR-GROUP/hpx/pull/5307
[1876] https://github.com/STEllAR-GROUP/hpx/pull/5304
[1877] https://github.com/STEllAR-GROUP/hpx/pull/5300
[1878] https://github.com/STEllAR-GROUP/hpx/pull/5298
[1879] https://github.com/STEllAR-GROUP/hpx/pull/5297
[1880] https://github.com/STEllAR-GROUP/hpx/pull/5295
[1881] https://github.com/STEllAR-GROUP/hpx/pull/5293
[1882] https://github.com/STEllAR-GROUP/hpx/pull/5292
[1883] https://github.com/STEllAR-GROUP/hpx/pull/5291
[1884] https://github.com/STEllAR-GROUP/hpx/pull/5290
[1885] https://github.com/STEllAR-GROUP/hpx/pull/5289
[1886] https://github.com/STEllAR-GROUP/hpx/pull/5288
[1887] https://github.com/STEllAR-GROUP/hpx/pull/5287
[1888] https://github.com/STEllAR-GROUP/hpx/pull/5285
[1889] https://github.com/STEllAR-GROUP/hpx/pull/5284

- PR #5282[1890] - Update Boost for daint 20.11 after update
- PR #5281[1891] - Fix Segmentation fault on `foreach_datapar_zipiter`
- PR #5280[1892] - Avoid modulo by zero in `counting_iterator` test
- PR #5279[1893] - Fix more GCC 10 deprecation warnings
- PR #5277[1894] - Small fixes and improvements to CUDA/MPI polling
- PR #5276[1895] - Fix typo in docs
- PR #5274[1896] - More P1897 algorithms
- PR #5273[1897] - Retry CDash submissions on failure
- PR #5272[1898] - Fix bogus deprecation warnings with GCC 10
- PR #5271[1899] - Correcting target ids for `symbol_namespace::iterate`
- PR #5268[1900] - Adding generic `require`, `require_concept`, and `query` properties
- PR #5267[1901] - Support annotations in `hpx::transform_reduce`
- PR #5266[1902] - Making late command line options available for local runtime
- PR #5265[1903] - Leverage `no_unique_address` for `member_pack`
- PR #5264[1904] - Adopt format in more places
- PR #5262[1905] - Install HPX in Rostam Jenkins jobs
- PR #5261[1906] - Limit Rostam Jenkins jobs to marvin partition temporarily
- PR #5260[1907] - Separate segmented algorithms for transform_reduce
- PR #5259[1908] - Making sure late command line options are recognized as configuration options
- PR #5258[1909] - Allow for HPX algorithms being invoked with std execution policies
- PR #5256[1910] - Separate segmented algorithms for transform
- PR #5255[1911] - Future/sender adapters
- PR #5254[1912] - Fixing datapar

[1890] https://github.com/STEllAR-GROUP/hpx/pull/5282
[1891] https://github.com/STEllAR-GROUP/hpx/pull/5281
[1892] https://github.com/STEllAR-GROUP/hpx/pull/5280
[1893] https://github.com/STEllAR-GROUP/hpx/pull/5279
[1894] https://github.com/STEllAR-GROUP/hpx/pull/5277
[1895] https://github.com/STEllAR-GROUP/hpx/pull/5276
[1896] https://github.com/STEllAR-GROUP/hpx/pull/5274
[1897] https://github.com/STEllAR-GROUP/hpx/pull/5273
[1898] https://github.com/STEllAR-GROUP/hpx/pull/5272
[1899] https://github.com/STEllAR-GROUP/hpx/pull/5271
[1900] https://github.com/STEllAR-GROUP/hpx/pull/5268
[1901] https://github.com/STEllAR-GROUP/hpx/pull/5267
[1902] https://github.com/STEllAR-GROUP/hpx/pull/5266
[1903] https://github.com/STEllAR-GROUP/hpx/pull/5265
[1904] https://github.com/STEllAR-GROUP/hpx/pull/5264
[1905] https://github.com/STEllAR-GROUP/hpx/pull/5262
[1906] https://github.com/STEllAR-GROUP/hpx/pull/5261
[1907] https://github.com/STEllAR-GROUP/hpx/pull/5260
[1908] https://github.com/STEllAR-GROUP/hpx/pull/5259
[1909] https://github.com/STEllAR-GROUP/hpx/pull/5258
[1910] https://github.com/STEllAR-GROUP/hpx/pull/5256
[1911] https://github.com/STEllAR-GROUP/hpx/pull/5255
[1912] https://github.com/STEllAR-GROUP/hpx/pull/5254

- PR #5253[1913] - Add utility to format ranges
- PR #5252[1914] - Remove uses of Boost.Bimap
- PR #5251[1915] - Banish `<iostream>` from library headers
- PR #5250[1916] - Try fixing vc circle ci
- PR #5249[1917] - Adding missing header
- PR #5248[1918] - Use old Piz Daint modules after upgrade
- PR #5247[1919] - Significantly speedup simple `for_each`, `for_loop`, and `transform`
- PR #5245[1920] - P1897 `operator|` overloads
- PR #5244[1921] - P1897 `when_all`
- PR #5243[1922] - Make sure `HPX_DEBUG` is set based on HPX's build type, not consuming project's build type
- PR #5242[1923] - Moving last files unrelated to parcel layer to modules
- PR #5240[1924] - change namespace for `transform_loop.hpp`
- PR #5238[1925] - Make sure annotations are used in the binary transform
- PR #5237[1926] - Add P1897 `just`, `just_on`, and `on` algorithms
- PR #5236[1927] - Add an example demonstrating the use of the `invoke_function_action` facility
- PR #5235[1928] - Attempting to fix datapar compilation issues
- PR #5234[1929] - Fix small typo in `--hpx:local` option description
- PR #5233[1930] - Only find Boost.Iostreams if required for plugins
- PR #5231[1931] - Sort printed config options
- PR #5230[1932] - Fix C++20 replace algo adaptation misses
- PR #5229[1933] - Remove leftover Boost include from `sync_wait.hpp`
- PR #5228[1934] - Print module name only if it has custom configuration settings
- PR #5227[1935] - Update .codespell_whitelist

1913 https://github.com/STEllAR-GROUP/hpx/pull/5253
1914 https://github.com/STEllAR-GROUP/hpx/pull/5252
1915 https://github.com/STEllAR-GROUP/hpx/pull/5251
1916 https://github.com/STEllAR-GROUP/hpx/pull/5250
1917 https://github.com/STEllAR-GROUP/hpx/pull/5249
1918 https://github.com/STEllAR-GROUP/hpx/pull/5248
1919 https://github.com/STEllAR-GROUP/hpx/pull/5247
1920 https://github.com/STEllAR-GROUP/hpx/pull/5245
1921 https://github.com/STEllAR-GROUP/hpx/pull/5244
1922 https://github.com/STEllAR-GROUP/hpx/pull/5243
1923 https://github.com/STEllAR-GROUP/hpx/pull/5242
1924 https://github.com/STEllAR-GROUP/hpx/pull/5240
1925 https://github.com/STEllAR-GROUP/hpx/pull/5238
1926 https://github.com/STEllAR-GROUP/hpx/pull/5237
1927 https://github.com/STEllAR-GROUP/hpx/pull/5236
1928 https://github.com/STEllAR-GROUP/hpx/pull/5235
1929 https://github.com/STEllAR-GROUP/hpx/pull/5234
1930 https://github.com/STEllAR-GROUP/hpx/pull/5233
1931 https://github.com/STEllAR-GROUP/hpx/pull/5231
1932 https://github.com/STEllAR-GROUP/hpx/pull/5230
1933 https://github.com/STEllAR-GROUP/hpx/pull/5229
1934 https://github.com/STEllAR-GROUP/hpx/pull/5228
1935 https://github.com/STEllAR-GROUP/hpx/pull/5227

- PR #5226[1936] - Use new docker image in all CircleCI steps
- PR #5225[1937] - Adapt reverse to C++20
- PR #5224[1938] - Separate segmented algorithms for `none_of`, `any_of` and `all_of`
- PR #5223[1939] - Fixing build system for ittnotify
- PR #5221[1940] - Moving LCO related files to modules
- PR #5220[1941] - Seperate segmented algorithms for `count` and `count_if`
- PR #5218[1942] - Seperate segmented algorithms for `adjacent_find`
- PR #5217[1943] - Add a HIP github action
- PR #5215[1944] - Update ROCm to 4.0.1 on Rostam
- PR #5214[1945] - Fix clang-format error in sender.hpp
- PR #5213[1946] - Removing ESSENTIAL option to the doc example
- PR #5212[1947] - Seperate segmented algorithms for `for_each_n`
- PR #5211[1948] - Minor adapted algos fixes
- PR #5210[1949] - Fixing `is_invocable` deprecation warnings
- PR #5209[1950] - Moving more files into modules (actions, components, init_runtime, etc.)
- PR #5208[1951] - Add examples and explanation on when `tag_fallback/priority` are useful
- PR #5207[1952] - Always define `HPX_COMPUTE_HOST_CODE` for host code
- PR #5206[1953] - Add formatting exceptions for libhpx to create_module_skeleton.py
- PR #5205[1954] - Moving all distribution policies into modules
- PR #5203[1955] - Move copy algorithms to `tag_fallback_invoke`
- PR #5202[1956] - Make `HPX_WITH_PSEUDO_DEPENDENCIES` a cache variable
- PR #5201[1957] - Replaced `tag_invoke` with `tag_fallback_invoke` for `adjacent_find` algorithm
- PR #5200[1958] - Moving files to (distributed) runtime module

[1936] https://github.com/STEllAR-GROUP/hpx/pull/5226
[1937] https://github.com/STEllAR-GROUP/hpx/pull/5225
[1938] https://github.com/STEllAR-GROUP/hpx/pull/5224
[1939] https://github.com/STEllAR-GROUP/hpx/pull/5223
[1940] https://github.com/STEllAR-GROUP/hpx/pull/5221
[1941] https://github.com/STEllAR-GROUP/hpx/pull/5220
[1942] https://github.com/STEllAR-GROUP/hpx/pull/5218
[1943] https://github.com/STEllAR-GROUP/hpx/pull/5217
[1944] https://github.com/STEllAR-GROUP/hpx/pull/5215
[1945] https://github.com/STEllAR-GROUP/hpx/pull/5214
[1946] https://github.com/STEllAR-GROUP/hpx/pull/5213
[1947] https://github.com/STEllAR-GROUP/hpx/pull/5212
[1948] https://github.com/STEllAR-GROUP/hpx/pull/5211
[1949] https://github.com/STEllAR-GROUP/hpx/pull/5210
[1950] https://github.com/STEllAR-GROUP/hpx/pull/5209
[1951] https://github.com/STEllAR-GROUP/hpx/pull/5208
[1952] https://github.com/STEllAR-GROUP/hpx/pull/5207
[1953] https://github.com/STEllAR-GROUP/hpx/pull/5206
[1954] https://github.com/STEllAR-GROUP/hpx/pull/5205
[1955] https://github.com/STEllAR-GROUP/hpx/pull/5203
[1956] https://github.com/STEllAR-GROUP/hpx/pull/5202
[1957] https://github.com/STEllAR-GROUP/hpx/pull/5201
[1958] https://github.com/STEllAR-GROUP/hpx/pull/5200

- PR #5199[1959] - Update ICC module name on Piz Daint Jenkins configuration
- PR #5198[1960] - Add doxygen documentation for thread_schedule_hint
- PR #5197[1961] - Attempt to fix compilation of context implementations with unity build enabled
- PR #5196[1962] - Re-enable component tests
- PR #5195[1963] - Moving files related to colocation logic
- PR #5194[1964] - Another attempt at fixing the Fedora 35 problem
- PR #5193[1965] - Components module
- PR #5192[1966] - Adapt `replace(_if)` to C++20
- PR #5190[1967] - Set compatibility headers by default to on
- PR #5188[1968] - Bump Boost minimum version to 1.71.0
- PR #5187[1969] - Force CMake to set the `-std=c++XX` flag
- PR #5186[1970] - Remove message to print .cu extension whenever .cu files are encountered
- PR #5185[1971] - Remove some minor unnecessary CMake options
- PR #5184[1972] - Remove some leftover `HPX_WITH_*_SCHEDULER` uses
- PR #5183[1973] - Remove dependency on boost/iterators/iterator_categories.hpp
- PR #5182[1974] - Fixing Fedora 35 for Power architectures
- PR #5181[1975] - Bump version number and tag post 1.6.0 release
- PR #5180[1976] - Fix htts_v2 tests linking
- PR #5179[1977] - Make sure `--hpx:local` command line option is respected with networking is off but distributed runtime is on
- PR #5177[1978] - Remove module cmake options
- PR #5176[1979] - Starting to separate segmented algorithms: `for_each`
- PR #5174[1980] - Don't run segmented algorithms twice on CircleCI
- PR #5173[1981] - Fetching APEX using cmake FetchContent

[1959] https://github.com/STEllAR-GROUP/hpx/pull/5199
[1960] https://github.com/STEllAR-GROUP/hpx/pull/5198
[1961] https://github.com/STEllAR-GROUP/hpx/pull/5197
[1962] https://github.com/STEllAR-GROUP/hpx/pull/5196
[1963] https://github.com/STEllAR-GROUP/hpx/pull/5195
[1964] https://github.com/STEllAR-GROUP/hpx/pull/5194
[1965] https://github.com/STEllAR-GROUP/hpx/pull/5193
[1966] https://github.com/STEllAR-GROUP/hpx/pull/5192
[1967] https://github.com/STEllAR-GROUP/hpx/pull/5190
[1968] https://github.com/STEllAR-GROUP/hpx/pull/5188
[1969] https://github.com/STEllAR-GROUP/hpx/pull/5187
[1970] https://github.com/STEllAR-GROUP/hpx/pull/5186
[1971] https://github.com/STEllAR-GROUP/hpx/pull/5185
[1972] https://github.com/STEllAR-GROUP/hpx/pull/5184
[1973] https://github.com/STEllAR-GROUP/hpx/pull/5183
[1974] https://github.com/STEllAR-GROUP/hpx/pull/5182
[1975] https://github.com/STEllAR-GROUP/hpx/pull/5181
[1976] https://github.com/STEllAR-GROUP/hpx/pull/5180
[1977] https://github.com/STEllAR-GROUP/hpx/pull/5179
[1978] https://github.com/STEllAR-GROUP/hpx/pull/5177
[1979] https://github.com/STEllAR-GROUP/hpx/pull/5176
[1980] https://github.com/STEllAR-GROUP/hpx/pull/5174
[1981] https://github.com/STEllAR-GROUP/hpx/pull/5173

- PR #5172[1982] - Add separate local-only entry point
- PR #5171[1983] - Remove `HPX_WITH_THREAD_SCHEDULERS` CMake option
- PR #5170[1984] - Add `HPX_WITH_PRECOMPILED_HEADERS` option
- PR #5166[1985] - Moving some action tests to modules
- PR #5165[1986] - Require cmake 3.17
- PR #5164[1987] - Move `thread_pool_suspension_helper` files to small utility module
- PR #5160[1988] - Adding checks ensuring modules are not cross-referenced from other module categories
- PR #5158[1989] - Replace boost::asio with standalone asio
- PR #5155[1990] - Allow logging when distributed runtime is off
- PR #5153[1991] - Components module
- PR #5152[1992] - Move more files to performance counter module
- PR #5150[1993] - Adapt `remove_copy(_if)` to C++20
- PR #5144[1994] - AGAS module
- PR #5125[1995] - Adapt `remove` and `remove_if` to C++20
- PR #5117[1996] - Attempt to fix segfaults assumed to be caused by `future_data` instances going out of scope.
- PR #5099[1997] - Allow mixing debug and release builds
- PR #5092[1998] - Replace spirit.qi with x3
- PR #5053[1999] - Add P0443r14 executor and a a few P1897 algorithms
- PR #5044[2000] - Add performance test in jenkins and reports

1982 https://github.com/STEllAR-GROUP/hpx/pull/5172
1983 https://github.com/STEllAR-GROUP/hpx/pull/5171
1984 https://github.com/STEllAR-GROUP/hpx/pull/5170
1985 https://github.com/STEllAR-GROUP/hpx/pull/5166
1986 https://github.com/STEllAR-GROUP/hpx/pull/5165
1987 https://github.com/STEllAR-GROUP/hpx/pull/5164
1988 https://github.com/STEllAR-GROUP/hpx/pull/5160
1989 https://github.com/STEllAR-GROUP/hpx/pull/5158
1990 https://github.com/STEllAR-GROUP/hpx/pull/5155
1991 https://github.com/STEllAR-GROUP/hpx/pull/5153
1992 https://github.com/STEllAR-GROUP/hpx/pull/5152
1993 https://github.com/STEllAR-GROUP/hpx/pull/5150
1994 https://github.com/STEllAR-GROUP/hpx/pull/5144
1995 https://github.com/STEllAR-GROUP/hpx/pull/5125
1996 https://github.com/STEllAR-GROUP/hpx/pull/5117
1997 https://github.com/STEllAR-GROUP/hpx/pull/5099
1998 https://github.com/STEllAR-GROUP/hpx/pull/5092
1999 https://github.com/STEllAR-GROUP/hpx/pull/5053
2000 https://github.com/STEllAR-GROUP/hpx/pull/5044

HPX V1.6.0 (Feb 17, 2021)

General changes

This release continues the focus on C++20 conformance with multiple new algorithms adapted to be C++20 conformant and becoming customization point objects (CPOs). We have also added experimental support for HIP, allowing previous CUDA features to now be compiled with hipcc and run on AMD GPUs.

- The following algorithms have been adapted to be C++20 conformant: `adjacent_find`, `includes`, `inplace_merge`, `is_heap`, `is_heap_until`, `is_partitioned`, `is_sorted`, `is_sorted_until`, `merge`, `set_difference`, `set_intersection`, `set_symmetric_difference`, `set_union`.
- Experimental HIP support can be enabled by compiling *HPX* with `hipcc`. All CUDA functionality in *HPX* can now be used with HIP. The HIP functionality is for the time being exposed through the same API as the CUDA functionality, i.e. no changes are required in user code. The CUDA, and now HIP, functionality is in the `hpx::cuda` namespace.
- We have added `partial_sort` based on Francisco Tapia's implementation.
- `hpx::init` and `hpx::start` gained new overloads taking an `hpx::init_params` struct in 1.5.0. All overloads not taking an `hpx::init_params` are now deprecated.
- We have added an experimental `fork_join_executor`. This executor can be used for OpenMP-style fork-join parallelism, where the latency of a parallel region is important for performance.
- The `parallel_executor` now uses a hierarchical spawning scheme for bulk execution, which improves data locality and performance.
- `hpx::dataflow` can now be used with executors that inject additional parameters into the call of the user-provided function.
- We have added experimental support for properties as proposed in P2220[2001]. Currently the only supported property is the scheduling hint on `parallel_executor`.
- *`hpx::util::annotated_function`* can now be passed a dynamically generated `std::string`.
- In moving functionality to new namespaces, old names have been deprecated. A deprecation warning will be issued if you are using deprecated functionality, with instructions on how to correct or ignore the warning.
- We have removed all support for C and Fortran from our build system.
- We have further reduced the use of Boost types within *HPX* (`boost::system::error_code` and `boost::detail::spinlock`).
- We have enabled more warnings in our CI builds (unused variables and unused typedefs).

Breaking changes

- hpxMP support has been completely removed.
- The `verbs` parcelport has been removed.
- The following compatibility options have been disabled by default: `HPX_WITH_ACTION_BASE_COMPATIBILITY`, `HPX_WITH_REGISTER_THREAD_COMPATIBILITY`, `HPX_WITH_PROMISE_ALIAS_COMPATIBILITY`, `HPX_WITH_UNSCOPED_ENUM_COMPATIBILITY`, `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY`, `HPX_WITH_EMBEDDED_THREAD_POOLS_COMPATIBILITY`, `HPX_WITH_THREAD_POOL_OS_EXECUTOR_COMPATIBILITY`, `HPX_WITH_THREAD_EXECUTORS_COMPATIBILITY`, `HPX_THREAD_AWARE_TIMER_COMPATIBILITY`, `HPX_WITH_POOL_EXECUTOR_COMPATIBILITY`. Unless noted here, the above functionalities do not come

[2001] https://wg21.link/p2220

with replacements. Unscoped enumerations have been replaced by scoped enumerations. Previously deprecated unscoped enumerations are disabled by `HPX_WITH_UNSCOPED_ENUM_COMPATIBILITY`. Newly deprecated unscoped enumerations have been given deprecation warnings and replaced by scoped enumerations. `hpx::promise` has been replaced with `hpx::distributed::promise`. `hpx::program_options` is a drop-in replacement for `boost::program_options`. `hpx::execution::parallel_executor` now has constructors which take a thread pool, covering the use case of `hpx::threads::executors::pool_executor`. A pool can be supplied with `hpx::resource::get_thread_pool`.

Closed issues

- Issue #5148[2002] - `runtime_support.hpp` does not work with newer cling
- Issue #5147[2003] - Wrong results with parallel reduce
- Issue #5129[2004] - Missing specialization for `std::hash<hpx::thread::id>`
- Issue #5126[2005] - Use `std::string` for task annotations
- Issue #5115[2006] - Don't expect hwloc to always report Cores
- Issue #5113[2007] - Handle threadmanager exceptions during startup
- Issue #5112[2008] - libatomic problems causing unexpected fails
- Issue #5089[2009] - Remove non-BSL files
- Issue #5088[2010] - Unwrapping problem
- Issue #5087[2011] - Remove hpxMP support
- Issue #5077[2012] - PAPI counters are not accessible when HPX is installed
- Issue #5075[2013] - Make the structs in all `iter_sent.hpp` lower case
- Issue #5067[2014] - Bug `string_util/split.hpp`
- Issue #5049[2015] - Change back the hipcc jenkins config to the fury partition on rostam
- Issue #5038[2016] - Not all examples link in the latest HPX master
- Issue #5035[2017] - Build with `HPX_WITH_EXAMPLES` fails
- Issue #5019[2018] - Broken help string for hpx
- Issue #5016[2019] - `hpx::parallel::fill` fails compiling

2002 https://github.com/STEllAR-GROUP/hpx/issues/5148
2003 https://github.com/STEllAR-GROUP/hpx/issues/5147
2004 https://github.com/STEllAR-GROUP/hpx/issues/5129
2005 https://github.com/STEllAR-GROUP/hpx/issues/5126
2006 https://github.com/STEllAR-GROUP/hpx/issues/5115
2007 https://github.com/STEllAR-GROUP/hpx/issues/5113
2008 https://github.com/STEllAR-GROUP/hpx/issues/5112
2009 https://github.com/STEllAR-GROUP/hpx/issues/5089
2010 https://github.com/STEllAR-GROUP/hpx/issues/5088
2011 https://github.com/STEllAR-GROUP/hpx/issues/5087
2012 https://github.com/STEllAR-GROUP/hpx/issues/5077
2013 https://github.com/STEllAR-GROUP/hpx/issues/5075
2014 https://github.com/STEllAR-GROUP/hpx/issues/5067
2015 https://github.com/STEllAR-GROUP/hpx/issues/5049
2016 https://github.com/STEllAR-GROUP/hpx/issues/5038
2017 https://github.com/STEllAR-GROUP/hpx/issues/5035
2018 https://github.com/STEllAR-GROUP/hpx/issues/5019
2019 https://github.com/STEllAR-GROUP/hpx/issues/5016

- Issue #5014[2020] - Rename all `.cc` to `.cpp` and `.hh` to `.hpp`
- Issue #4988[2021] - MPI is not finalized if running with only one locality
- Issue #4978[2022] - Change feature test macros to expand to zero/one
- Issue #4949[2023] - Crash when not enabling TCP parcelport
- Issue #4933[2024] - Improve test coverage for unused variable warnings etc.
- Issue #4878[2025] - HPX mpi async might call `MPI_FINALIZE` before app calls it
- Issue #4127[2026] - Local runtime entry-points

Closed pull requests

- PR #5178[2027] - Fix parallel `remove/remove_copy/transform` namespace references in docs
- PR #5169[2028] - Attempt to get Piz Daint jenkins setup running after maintenance
- PR #5168[2029] - Remove include of itself
- PR #5167[2030] - Fixing deprecation warnings that slipped through the net
- PR #5159[2031] - Update APEX tag to 2.3.1
- PR #5154[2032] - Splitting unit tests on circleci to avoid timeouts
- PR #5151[2033] - Use C++20 on `clang-newest` Jenkins CI configuration
- PR #5149[2034] - Rename `'module'` symbols to avoid keyword conflict
- PR #5145[2035] - Adjust handling of CUDA/HIP options in CMake
- PR #5142[2036] - Store annotated_function annotations as `std::strings`
- PR #5140[2037] - Scheduler mode
- PR #5139[2038] - Fix path problem in pre-commit hook, add summary commit line
- PR #5138[2039] - Add program options variable map to resource partitioner init
- PR #5137[2040] - Remove the use of `boost::throw_exception`
- PR #5136[2041] - Make sure codespell checks run on CircleCI

2020 https://github.com/STEllAR-GROUP/hpx/issues/5014
2021 https://github.com/STEllAR-GROUP/hpx/issues/4988
2022 https://github.com/STEllAR-GROUP/hpx/issues/4978
2023 https://github.com/STEllAR-GROUP/hpx/issues/4949
2024 https://github.com/STEllAR-GROUP/hpx/issues/4933
2025 https://github.com/STEllAR-GROUP/hpx/issues/4878
2026 https://github.com/STEllAR-GROUP/hpx/issues/4127
2027 https://github.com/STEllAR-GROUP/hpx/pull/5178
2028 https://github.com/STEllAR-GROUP/hpx/pull/5169
2029 https://github.com/STEllAR-GROUP/hpx/pull/5168
2030 https://github.com/STEllAR-GROUP/hpx/pull/5167
2031 https://github.com/STEllAR-GROUP/hpx/pull/5159
2032 https://github.com/STEllAR-GROUP/hpx/pull/5154
2033 https://github.com/STEllAR-GROUP/hpx/pull/5151
2034 https://github.com/STEllAR-GROUP/hpx/pull/5149
2035 https://github.com/STEllAR-GROUP/hpx/pull/5145
2036 https://github.com/STEllAR-GROUP/hpx/pull/5142
2037 https://github.com/STEllAR-GROUP/hpx/pull/5140
2038 https://github.com/STEllAR-GROUP/hpx/pull/5139
2039 https://github.com/STEllAR-GROUP/hpx/pull/5138
2040 https://github.com/STEllAR-GROUP/hpx/pull/5137
2041 https://github.com/STEllAR-GROUP/hpx/pull/5136

- PR #5132[2042] - Fixing spelling errors
- PR #5131[2043] - Mark `counting_iterator` member functions as `HPX_HOST_DEVICE`
- PR #5130[2044] - Adding specialization for `std::hash<hpx::thread::id>`
- PR #5128[2045] - Fixing environment handling for FreeBSD
- PR #5127[2046] - Fix typo in fibonacci documentation
- PR #5123[2047] - Reduce vector sizes in partial sort benchmarks when running in debug mode
- PR #5122[2048] - Making sure exceptions during runtime initialization are correctly reported
- PR #5121[2049] - Working around hwloc limitation on certain platforms
- PR #5120[2050] - Fixing compatibility warnings in `hpx::transform` implementation
- PR #5119[2051] - Use `sequential_find` and friends from separate detail header
- PR #5116[2052] - Fix compilation with timer pool off
- PR #5114[2053] - Fix 5112 - make sure libatomic is used when needed
- PR #5109[2054] - Remove default runtime mode argument from init overload, again
- PR #5108[2055] - Refactor `iter_sent.hpp` to make structs lowercase
- PR #5107[2056] - Relax `dataflow` internals
- PR #5106[2057] - Change initialization of property CPOs to satisfy older nvcc versions
- PR #5104[2058] - Fix regeneration of two files that trigger unnecessary rebuilds
- PR #5103[2059] - Remove default runtime mode argument from start/init overloads
- PR #5102[2060] - Untie deprecated thread enums from the CMake option
- PR #5101[2061] - Update APEX tag for 1.6.0
- PR #5100[2062] - Bump minimum required Boost version to 1.66 and update CI configurations
- PR #5098[2063] - Minor fixes to public API listing
- PR #5097[2064] - Remove hpxMP support

2042 https://github.com/STEllAR-GROUP/hpx/pull/5132
2043 https://github.com/STEllAR-GROUP/hpx/pull/5131
2044 https://github.com/STEllAR-GROUP/hpx/pull/5130
2045 https://github.com/STEllAR-GROUP/hpx/pull/5128
2046 https://github.com/STEllAR-GROUP/hpx/pull/5127
2047 https://github.com/STEllAR-GROUP/hpx/pull/5123
2048 https://github.com/STEllAR-GROUP/hpx/pull/5122
2049 https://github.com/STEllAR-GROUP/hpx/pull/5121
2050 https://github.com/STEllAR-GROUP/hpx/pull/5120
2051 https://github.com/STEllAR-GROUP/hpx/pull/5119
2052 https://github.com/STEllAR-GROUP/hpx/pull/5116
2053 https://github.com/STEllAR-GROUP/hpx/pull/5114
2054 https://github.com/STEllAR-GROUP/hpx/pull/5109
2055 https://github.com/STEllAR-GROUP/hpx/pull/5108
2056 https://github.com/STEllAR-GROUP/hpx/pull/5107
2057 https://github.com/STEllAR-GROUP/hpx/pull/5106
2058 https://github.com/STEllAR-GROUP/hpx/pull/5104
2059 https://github.com/STEllAR-GROUP/hpx/pull/5103
2060 https://github.com/STEllAR-GROUP/hpx/pull/5102
2061 https://github.com/STEllAR-GROUP/hpx/pull/5101
2062 https://github.com/STEllAR-GROUP/hpx/pull/5100
2063 https://github.com/STEllAR-GROUP/hpx/pull/5098
2064 https://github.com/STEllAR-GROUP/hpx/pull/5097

- PR #5096[2065] - Remove fractals examples
- PR #5095[2066] - Use all AMD nodes again on rostam
- PR #5094[2067] - Attempt to remove macOS workaround for GH actions environment
- PR #5093[2068] - Remove verbs parcelport
- PR #5091[2069] - Avoid moving from lvalues
- PR #5090[2070] - Adopt C++20 `std::endian`
- PR #5085[2071] - Update daint CI to use Boost 1.75.0
- PR #5084[2072] - Disable compatibility options for 1.6.0 release
- PR #5083[2073] - Remove duplicated call to the `limiting_executor` in `future_overhead` test
- PR #5079[2074] - Add checks to make sure that MPI/CUDA polling is enabled/not disabled too early
- PR #5078[2075] - Add install lib directory to list of component search paths
- PR #5076[2076] - Fix a typo in the jenkins `clang-newest` cmake config
- PR #5074[2077] - Fixing warnings generated by MSVC
- PR #5073[2078] - Allow using noncopyable types with unwrapping
- PR #5072[2079] - Fix `is_convertible` args in `result_types`
- PR #5071[2080] - Fix unused parameters
- PR #5070[2081] - Fix unused variables warnings in hipcc
- PR #5069[2082] - Add support for sentinels to `adjacent_find`
- PR #5068[2083] - Fix string split function
- PR #5066[2084] - Adapt `search` to C++20 and Range TS
- PR #5065[2085] - Fix `hpx::range::adjacent_find` doxygen function signatures
- PR #5064[2086] - Refactor runtime configuration, command line handling, and resource partitioner
- PR #5063[2087] - Limit the device code guards to the distributed parts of the `future_overhead` bench

2065 https://github.com/STEllAR-GROUP/hpx/pull/5096
2066 https://github.com/STEllAR-GROUP/hpx/pull/5095
2067 https://github.com/STEllAR-GROUP/hpx/pull/5094
2068 https://github.com/STEllAR-GROUP/hpx/pull/5093
2069 https://github.com/STEllAR-GROUP/hpx/pull/5091
2070 https://github.com/STEllAR-GROUP/hpx/pull/5090
2071 https://github.com/STEllAR-GROUP/hpx/pull/5085
2072 https://github.com/STEllAR-GROUP/hpx/pull/5084
2073 https://github.com/STEllAR-GROUP/hpx/pull/5083
2074 https://github.com/STEllAR-GROUP/hpx/pull/5079
2075 https://github.com/STEllAR-GROUP/hpx/pull/5078
2076 https://github.com/STEllAR-GROUP/hpx/pull/5076
2077 https://github.com/STEllAR-GROUP/hpx/pull/5074
2078 https://github.com/STEllAR-GROUP/hpx/pull/5073
2079 https://github.com/STEllAR-GROUP/hpx/pull/5072
2080 https://github.com/STEllAR-GROUP/hpx/pull/5071
2081 https://github.com/STEllAR-GROUP/hpx/pull/5070
2082 https://github.com/STEllAR-GROUP/hpx/pull/5069
2083 https://github.com/STEllAR-GROUP/hpx/pull/5068
2084 https://github.com/STEllAR-GROUP/hpx/pull/5066
2085 https://github.com/STEllAR-GROUP/hpx/pull/5065
2086 https://github.com/STEllAR-GROUP/hpx/pull/5064
2087 https://github.com/STEllAR-GROUP/hpx/pull/5063

- PR #5061[2088] - Remove hipcc guards in examples and tests
- PR #5060[2089] - Fix deprecation warnings generated by msvc
- PR #5059[2090] - Add warning about suspending/resuming the runtime in multi-locality scenarios
- PR #5057[2091] - Fix unused variable warnings
- PR #5056[2092] - Fix `hpx::util::get`
- PR #5055[2093] - Remove hipcc guards
- PR #5054[2094] - Fix typo
- PR #5051[2095] - Adapt transform to C++20
- PR #5050[2096] - Replace old init overloads in tests and examples
- PR #5048[2097] - Limit jenkins hipcc to the reno node
- PR #5047[2098] - Limit cuda jenkins run to nodes with exclusively Nvidia GPUs
- PR #5046[2099] - Convert thread and future enums to class enums
- PR #5043[2100] - Improve `hpxrun.py` for Phylanx
- PR #5042[2101] - Add missing header to partial sort test
- PR #5041[2102] - Adding Francisco Tapia's implementation of `partial_sort`
- PR #5040[2103] - Remove generated headers left behind from a previous configuration
- PR #5039[2104] - Fix GCC 10 release builds
- PR #5037[2105] - Add `is_invocable` typedefs to top-level `hpx` namespace and public API list
- PR #5036[2106] - Deprecate `hpx::util::decay` in favor of `std::decay`
- PR #5034[2107] - Use versioned container image on CircleCI
- PR #5033[2108] - Implement P2220 properties module
- PR #5032[2109] - Do codespell comparison only on files changed from common ancestor
- PR #5031[2110] - Moving traits files to `actions_base`

2088 https://github.com/STEllAR-GROUP/hpx/pull/5061
2089 https://github.com/STEllAR-GROUP/hpx/pull/5060
2090 https://github.com/STEllAR-GROUP/hpx/pull/5059
2091 https://github.com/STEllAR-GROUP/hpx/pull/5057
2092 https://github.com/STEllAR-GROUP/hpx/pull/5056
2093 https://github.com/STEllAR-GROUP/hpx/pull/5055
2094 https://github.com/STEllAR-GROUP/hpx/pull/5054
2095 https://github.com/STEllAR-GROUP/hpx/pull/5051
2096 https://github.com/STEllAR-GROUP/hpx/pull/5050
2097 https://github.com/STEllAR-GROUP/hpx/pull/5048
2098 https://github.com/STEllAR-GROUP/hpx/pull/5047
2099 https://github.com/STEllAR-GROUP/hpx/pull/5046
2100 https://github.com/STEllAR-GROUP/hpx/pull/5043
2101 https://github.com/STEllAR-GROUP/hpx/pull/5042
2102 https://github.com/STEllAR-GROUP/hpx/pull/5041
2103 https://github.com/STEllAR-GROUP/hpx/pull/5040
2104 https://github.com/STEllAR-GROUP/hpx/pull/5039
2105 https://github.com/STEllAR-GROUP/hpx/pull/5037
2106 https://github.com/STEllAR-GROUP/hpx/pull/5036
2107 https://github.com/STEllAR-GROUP/hpx/pull/5034
2108 https://github.com/STEllAR-GROUP/hpx/pull/5033
2109 https://github.com/STEllAR-GROUP/hpx/pull/5032
2110 https://github.com/STEllAR-GROUP/hpx/pull/5031

- PR #5030[2111] - Add codespell version print in circleci
- PR #5029[2112] - Work around problems in GitHub actions macOS builder
- PR #5028[2113] - Moving move files to naming and naming_base
- PR #5027[2114] - Lessen constraints on certain algorithm arguments
- PR #5025[2115] - Adapt `is_sorted` and `is_sorted_until` to C++20
- PR #5024[2116] - Moving `naming_base` to full modules
- PR #5022[2117] - Remove C language from `CMakeLists.txt`
- PR #5021[2118] - Warn about unused arguments given to `add_hpx_module`
- PR #5020[2119] - Fixing help string
- PR #5018[2120] - Update CSCS jenkins configuration to clang 11
- PR #5017[2121] - Fixing broken backwards compatibility for `hpx::parallel::fill`
- PR #5015[2122] - Detect if generated global header conflicts with explicitly listed module headers
- PR #5012[2123] - Properly reset pointer tracking data in `output_archive`
- PR #5011[2124] - Inspect command line tweaks
- PR #5010[2125] - Creating AGAS module
- PR #5009[2126] - Replace `boost::system::error_code` with `std::error_code`
- PR #5008[2127] - Replace uses of `boost::detail::spinlock`
- PR #5007[2128] - Bump minimal Boost version to 1.65.0
- PR #5006[2129] - Adapt is_partitioned to C++20
- PR #5005[2130] - Making sure `reduce_by_key` compiles again
- PR #5004[2131] - Fixing template specializations that make extra archive data types unique across module boundaries
- PR #5003[2132] - Relax `dataflow` argument constraints
- PR #5001[2133] - Add <`random`> inspect check

2111 https://github.com/STEllAR-GROUP/hpx/pull/5030
2112 https://github.com/STEllAR-GROUP/hpx/pull/5029
2113 https://github.com/STEllAR-GROUP/hpx/pull/5028
2114 https://github.com/STEllAR-GROUP/hpx/pull/5027
2115 https://github.com/STEllAR-GROUP/hpx/pull/5025
2116 https://github.com/STEllAR-GROUP/hpx/pull/5024
2117 https://github.com/STEllAR-GROUP/hpx/pull/5022
2118 https://github.com/STEllAR-GROUP/hpx/pull/5021
2119 https://github.com/STEllAR-GROUP/hpx/pull/5020
2120 https://github.com/STEllAR-GROUP/hpx/pull/5018
2121 https://github.com/STEllAR-GROUP/hpx/pull/5017
2122 https://github.com/STEllAR-GROUP/hpx/pull/5015
2123 https://github.com/STEllAR-GROUP/hpx/pull/5012
2124 https://github.com/STEllAR-GROUP/hpx/pull/5011
2125 https://github.com/STEllAR-GROUP/hpx/pull/5010
2126 https://github.com/STEllAR-GROUP/hpx/pull/5009
2127 https://github.com/STEllAR-GROUP/hpx/pull/5008
2128 https://github.com/STEllAR-GROUP/hpx/pull/5007
2129 https://github.com/STEllAR-GROUP/hpx/pull/5006
2130 https://github.com/STEllAR-GROUP/hpx/pull/5005
2131 https://github.com/STEllAR-GROUP/hpx/pull/5004
2132 https://github.com/STEllAR-GROUP/hpx/pull/5003
2133 https://github.com/STEllAR-GROUP/hpx/pull/5001

- PR #4999[2134] - Attempt to fix MacOS Github action error
- PR #4997[2135] - Fix unused variable and typedef warnings
- PR #4996[2136] - Adapt `adjacent_find` to C++20
- PR #4995[2137] - Test all schedulers in `cross_pool_injection` test except `shared_priority_queue_scheduler`
- PR #4993[2138] - Fix deprecation warnings
- PR #4991[2139] - Avoid unnecessarily including entire modules
- PR #4990[2140] - Fixing some warnings from HPX complaining about use of obsolete types
- PR #4989[2141] - add a *destroy* trait for ParcelPort plugins
- PR #4986[2142] - Remove serialization to functional module dependency
- PR #4985[2143] - Compatibility header generation
- PR #4980[2144] - Add ranges overloads to `for_loop` (and variants)
- PR #4979[2145] - Actually enable unity builds on Jenkins
- PR #4977[2146] - Cleaning up `debug::print` functionalities
- PR #4976[2147] - Remove indirection layer in `at_index_impl`
- PR #4975[2148] - Remove indirection layer in `at_index_impl`
- PR #4973[2149] - Avoid warnings/errors for older gcc complaining about multi-line comments
- PR #4970[2150] - Making set algorithms conform to C++20
- PR #4969[2151] - Moving `is_execution_policy` and friends into namespace `hpx`
- PR #4968[2152] - Enable deprecation warnings for 1.6.0 and move `any` functionality to hpx namespace
- PR #4967[2153] - Define deprecation macros conditionally
- PR #4966[2154] - Add `clang-format` and `cmake-format` version prints
- PR #4965[2155] - Making `is_heap` and `is_heap_until` conforming to C++20
- PR #4964[2156] - Adding parallel `make_heap`

[2134] https://github.com/STEllAR-GROUP/hpx/pull/4999
[2135] https://github.com/STEllAR-GROUP/hpx/pull/4997
[2136] https://github.com/STEllAR-GROUP/hpx/pull/4996
[2137] https://github.com/STEllAR-GROUP/hpx/pull/4995
[2138] https://github.com/STEllAR-GROUP/hpx/pull/4993
[2139] https://github.com/STEllAR-GROUP/hpx/pull/4991
[2140] https://github.com/STEllAR-GROUP/hpx/pull/4990
[2141] https://github.com/STEllAR-GROUP/hpx/pull/4989
[2142] https://github.com/STEllAR-GROUP/hpx/pull/4986
[2143] https://github.com/STEllAR-GROUP/hpx/pull/4985
[2144] https://github.com/STEllAR-GROUP/hpx/pull/4980
[2145] https://github.com/STEllAR-GROUP/hpx/pull/4979
[2146] https://github.com/STEllAR-GROUP/hpx/pull/4977
[2147] https://github.com/STEllAR-GROUP/hpx/pull/4976
[2148] https://github.com/STEllAR-GROUP/hpx/pull/4975
[2149] https://github.com/STEllAR-GROUP/hpx/pull/4973
[2150] https://github.com/STEllAR-GROUP/hpx/pull/4970
[2151] https://github.com/STEllAR-GROUP/hpx/pull/4969
[2152] https://github.com/STEllAR-GROUP/hpx/pull/4968
[2153] https://github.com/STEllAR-GROUP/hpx/pull/4967
[2154] https://github.com/STEllAR-GROUP/hpx/pull/4966
[2155] https://github.com/STEllAR-GROUP/hpx/pull/4965
[2156] https://github.com/STEllAR-GROUP/hpx/pull/4964

- PR #4962[2157] - Fix external timer function pointer exports
- PR #4960[2158] - Fixing folder names for module tests and examples
- PR #4959[2159] - Adding communications set
- PR #4958[2160] - Deprecate tuple and timing functionality `in hpx::util`
- PR #4957[2161] - Fixing unity build option for parcelports
- PR #4953[2162] - Fixing MSVC problems after recent restructurings
- PR #4952[2163] - Make `parallel_executor` use `thread_pool_executor` spawning mechanism
- PR #4948[2164] - Clean up old artifacts better and more aggressively on Jenkins
- PR #4947[2165] - Add HIP support for AMD GPUs
- PR #4945[2166] - Enable `HPX_WITH_UNITY_BUILD` option on one of the Jenkins configurations
- PR #4943[2167] - Move public `hpx::parallel::execution` functionality to hpx::execution
- PR #4938[2168] - Post release cleanup
- PR #4858[2169] - Extending resilience APIs to support distributed invocations
- PR #4744[2170] - Fork-join executor
- PR #4665[2171] - Implementing sender, receiver, and `operation_state` concepts in terms of P0443r13
- PR #4649[2172] - Split libhpx into multiple libraries
- PR #4642[2173] - Implementing `operation_state` concept in terms of P0443r13
- PR #4640[2174] - Implementing receiver concept in terms of P0443r13
- PR #4622[2175] - Sanitizer fixes

[2157] https://github.com/STEllAR-GROUP/hpx/pull/4962
[2158] https://github.com/STEllAR-GROUP/hpx/pull/4960
[2159] https://github.com/STEllAR-GROUP/hpx/pull/4959
[2160] https://github.com/STEllAR-GROUP/hpx/pull/4958
[2161] https://github.com/STEllAR-GROUP/hpx/pull/4957
[2162] https://github.com/STEllAR-GROUP/hpx/pull/4953
[2163] https://github.com/STEllAR-GROUP/hpx/pull/4952
[2164] https://github.com/STEllAR-GROUP/hpx/pull/4948
[2165] https://github.com/STEllAR-GROUP/hpx/pull/4947
[2166] https://github.com/STEllAR-GROUP/hpx/pull/4945
[2167] https://github.com/STEllAR-GROUP/hpx/pull/4943
[2168] https://github.com/STEllAR-GROUP/hpx/pull/4938
[2169] https://github.com/STEllAR-GROUP/hpx/pull/4858
[2170] https://github.com/STEllAR-GROUP/hpx/pull/4744
[2171] https://github.com/STEllAR-GROUP/hpx/pull/4665
[2172] https://github.com/STEllAR-GROUP/hpx/pull/4649
[2173] https://github.com/STEllAR-GROUP/hpx/pull/4642
[2174] https://github.com/STEllAR-GROUP/hpx/pull/4640
[2175] https://github.com/STEllAR-GROUP/hpx/pull/4622

HPX V1.5.1 (Sep 30, 2020)

General changes

This is a patch release. It contains the following changes:

- Remove restriction on suspending runtime with multiple localities, users are now responsible for synchronizing work between localities before suspending.
- Fixes several compilation problems and warnings.
- Adds notes in the documentation explaining how to cite HPX.

Closed issues

- Issue #4971[2176] - Parallel sort fails to compile with C++20
- Issue #4950[2177] - Build with *HPX_WITH_PARCELPORT_ACTION_COUNTERS ON* fails
- Issue #4940[2178] - Codespell report for "HPX" (on fossies.org)
- Issue #4937[2179] - Allow suspension of runtime for multiple localities

Closed pull requests

- PR #4982[2180] - Add page about citing HPX to documentation
- PR #4981[2181] - Adding the missing include
- PR #4974[2182] - Remove leftover format export hack
- PR #4972[2183] - Removing use of get_temporary_buffer and return_temporary_buffer
- PR #4963[2184] - Renaming files to avoid warnings from the vs build system
- PR #4951[2185] - Fixing build if HPX_WITH_PARCELPORT_ACTION_COUNTERS=On
- PR #4946[2186] - Allow suspension on multiple localities
- PR #4944[2187] - Fix typos reported by fossies codespell report
- PR #4941[2188] - Adding some explanation to README about how to cite HPX
- PR #4939[2189] - Small changes

[2176] https://github.com/STEllAR-GROUP/hpx/issues/4971
[2177] https://github.com/STEllAR-GROUP/hpx/issues/4950
[2178] https://github.com/STEllAR-GROUP/hpx/issues/4940
[2179] https://github.com/STEllAR-GROUP/hpx/issues/4937
[2180] https://github.com/STEllAR-GROUP/hpx/pull/4982
[2181] https://github.com/STEllAR-GROUP/hpx/pull/4981
[2182] https://github.com/STEllAR-GROUP/hpx/pull/4974
[2183] https://github.com/STEllAR-GROUP/hpx/pull/4972
[2184] https://github.com/STEllAR-GROUP/hpx/pull/4963
[2185] https://github.com/STEllAR-GROUP/hpx/pull/4951
[2186] https://github.com/STEllAR-GROUP/hpx/pull/4946
[2187] https://github.com/STEllAR-GROUP/hpx/pull/4944
[2188] https://github.com/STEllAR-GROUP/hpx/pull/4941
[2189] https://github.com/STEllAR-GROUP/hpx/pull/4939

HPX V1.5.0 (Sep 02, 2020)

General changes

The main focus of this release is on APIs and C++20 conformance. We have added many new C++20 features and adapted multiple algorithms to be fully C++20 conformant. As part of the modularization we have begun specifying the public API of *HPX* in terms of headers and functionality, and aligning it more closely to the C++ standard. All non-distributed modules are now in place, along with an experimental option to completely disable distributed features in *HPX*. We have also added experimental asynchronous MPI and CUDA executors. Lastly this release introduces CMake targets for depending projects, performance improvements, and many bug fixes.

- We have added the C++20 features `hpx::jthread` and `hpx::stop_token`. `hpx::condition_variable_any` now exposes new functions supporting `hpx::stop_token`.
- We have added `hpx::stable_sort` based on Francisco Tapia's implementation.
- We have adapted existing synchronization primitives to be fully conformant C++20: `hpx::barrier`, `hpx::latch`, `hpx::counting_semaphore`, and `hpx::binary_semaphore`.
- We have started using customization point objects (CPOs) to make the corresponding algorithms fully conformant to C++20 as well as to make algorithm extension easier for the user. `all_of`/`any_of`/`none_of`, `copy`, `count`, `destroy`, `equal`, `fill`, `find`, `for_each`, `generate`, `mismatch`, `move`, `reduce`, `transform_reduce` are using those CPOs (all in namespace `hpx`). We also have adapted their corresponding `hpx::ranges` versions to be conforming to C++20 in this release.
- We have adapted support for `co_await` to C++20, in addition to `hpx::future` it now also supports `hpx::shared_future`. We have also added allocator support for futures returned by `co_return`. It is no longer in the `experimental` namespace.
- We added serialization support for `std::variant` and `std::tuple`.
- `result_of` and `is_callable` are now deprecated and replaced by `invoke_result` and `is_invocable` to conform to C++20.
- We continued with the modularization, making it easier for us to add the new experimental `HPX_WITH_DISTRIBUTED_RUNTIME` CMake option (see below) . An significant amount of headers have been deprecated. We adapted the namespaces and headers we could to be closer to the standard ones (*Public API*). Depending code should still compile, however warnings are now generated instructing to change the include statements accordingly.
- It is now possible to have a basic CUDA support including a helper function to get a future from a CUDA stream and target handling. They are available under the `hpx::cuda::experimental` namespace and they can be enabled with the `-DHPX_WITH_ASYNC_CUDA=ON` CMake option.
- We added a new `hpx::mpi::experimental` namespace for getting futures from an asynchronous MPI call and a new minimal MPI executor `hpx::mpi::experimental::executor`. These can be enabled with the `-DHPX_WITH_ASYNC_MPI=On` CMake option.
- A polymorphic executor has been implemented to reduce compile times as a function accepting executors can potentially be instantiated only once instead of multiple times with different executors. It accepts the function signature as a template argument. It needs to be constructed from any other executor. Please note, that the function signatures that can be scheduled using `then_execute`, `bulk_sync_execute`, `bulk_async_execute` and `bulk_then_execute` are slightly different (See the comment in PR #4514[2190] for more details).
- The underlying executor of `block_executor` has been updated to a newer one.
- We have added a parameter to `auto_chunk_size` to control the amount of iterations to measure.

[2190] https://github.com/STEllAR-GROUP/hpx/pull/4514

- All executor parameter hooks can now be exposed through the executor itself. This will allow to deprecate the `.with()` functionality on execution policies in the future. This is also a first step towards simplifying our executor APIs in preparation for the upcoming C++23 executors (senders/receivers).
- We have moved all of the existing APIs related to resiliency into the namespace `hpx::resiliency::experimental`. Please note this is a breaking change without backwards-compatibility option. We have converted all of those APIs to be based on customization point objects. Two new executors have been added to enable easy integration of the existing resiliency features with other facilities (like the parallel algorithms): `replay_executor` and `replicate_executor`.
- We have added performance counters type information (`aggregating`, `monotonically increasing`, `average count`, `average timer`, etc.).
- HPX threads are now re-scheduled on the same worker thread they were suspended on to avoid cache misses from moving from one thread to the other. This behavior doesn't prevent the thread from being stolen, however.
- We have added a new configuration option `hpx.exception_verbosity` to allow to control the level of verbosity of the exceptions (3 levels available).
- `broadcast_to`, `broadcast_from`, `scatter_to` and `scatter_from` have been added to the collectives, modernization of `gather_here` and `gather_there` with futures taken by rvalue references. See the breaking change on `all_to_all` in the next section. None of the collectives need supporting macros anymore (e.g. specifying the data types used for a collective operation using `HPX_REGISTER_ALLGATHER` and similar is not needed anymore).
- New API functions have been added: a) to get the number of cores which are idle (`hpx::get_idle_core_count`) and b) returning a bitmask representing the currently idle cores (`hpx::get_idle_core_mask`).
- We have added an experimental option to only enable the local runtime, you can disable the distributed runtime with `HPX_WITH_DISTRIBUTED_RUNTIME=OFF`. You can also enable the local runtime by using the `--hpx:local` runtime option.
- We fixed task annotations for actions.
- The alias `hpx::promise` to `hpx::lcos::promise` is now deprecated. You can use `hpx::lcos::promise` directly instead. `hpx::promise` will refer to the local-only promise in the future.
- We have added a `prepare_checkpoint` API function that calculates the amount of necessary buffer space for a particular set of arguments checkpointed.
- We have added `hpx::upgrade_lock` and `hpx::upgrade_to_unique_lock`, which make `hpx::shared_mutex` (and similar) usable in more flexible ways.
- We have changed the CMake targets exposed to the user, it now includes `HPX::hpx`, `HPX::wrap_main` (`int main` as the first *HPX* thread of the application, see *Starting the HPX runtime*), `HPX::plugin`, `HPX::component`. The CMake variables `HPX_INCLUDE_DIRS` and `HPX_LIBRARIES` are deprecated and will be removed in a future release, you should now link directly to the `HPX::hpx` CMake target.
- A new example is demonstrating how to create and use a wrapping executor (`quickstart/executor_with_thread_hooks.cpp`)
- A new example is demonstrating how to disable thread stealing during the execution of parallel algorithms (`quickstart/disable_thread_stealing_executor.cpp`)
- We now require for our CMake build system configuration files to be formatted using cmake-format.
- We have removed more dependencies on various Boost libraries.
- We have added an experimental option enabling unity builds of HPX using the `-DHPX_WITH_UNITY_BUILD=On` CMake option.
- Many bug fixes.

Breaking changes

- *HPX* now requires a C++14 capable compiler. We have set the *HPX* C++ standard automatically to C++14 and if it needs to be set explicitly, it should be specified through the `CMAKE_CXX_STANDARD` setting as mandated by CMake. The `HPX_WITH_CXX*` variables are now deprecated and will be removed in the future.
- Building and using HPX is now supported only when using CMake V3.13 or later, Boost V1.64 or newer, and when compiling with clang V5, gcc V7, or VS2019, or later. Other compilers might still work but have not been tested thoroughly.
- We have added a `hpx::init_params` struct to pass parameters for *HPX* initialization e.g. the resource partitioner callback to initialize thread pools (*Using the resource partitioner*).
- The `all_to_all` algorithm is renamed to `all_gather`, and the new `all_to_all` algorithm is not compatible with the old one.
- We have moved all of the existing APIs related to resiliency into the namespace `hpx::resiliency::experimental`.

Closed issues

- Issue #4918[2191] - Rename distributed_executors module
- Issue #4900[2192] - Adding JOSS status badge to README
- Issue #4897[2193] - Compiler warning, deprecated header used by HPX itself
- Issue #4886[2194] - A future bound to an action executing on a different locality doesn't capture exception state
- Issue #4880[2195] - Undefined reference to main build error when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- Issue #4877[2196] - hpx_main might not able to start hpx runtime properly
- Issue #4850[2197] - Issues creating templated component
- Issue #4829[2198] - Spack package & HPX_WITH_GENERIC_CONTEXT_COROUTINES
- Issue #4820[2199] - PAPI counters don't work
- Issue #4818[2200] - HPX can't be used with IO pool turned off
- Issue #4816[2201] - Build of HPX fails when find_package(Boost) is called before FetchContent_MakeAvailable(hpx)
- Issue #4813[2202] - HPX MPI Future failed
- Issue #4811[2203] - Remove HPX::hpx_no_wrap_main target before 1.5.0 release
- Issue #4810[2204] - In hpx::for_each::invoke_projected the hpx::util::decay is misguided

[2191] https://github.com/STEllAR-GROUP/hpx/issues/4918
[2192] https://github.com/STEllAR-GROUP/hpx/issues/4900
[2193] https://github.com/STEllAR-GROUP/hpx/issues/4897
[2194] https://github.com/STEllAR-GROUP/hpx/issues/4886
[2195] https://github.com/STEllAR-GROUP/hpx/issues/4880
[2196] https://github.com/STEllAR-GROUP/hpx/issues/4877
[2197] https://github.com/STEllAR-GROUP/hpx/issues/4850
[2198] https://github.com/STEllAR-GROUP/hpx/issues/4829
[2199] https://github.com/STEllAR-GROUP/hpx/issues/4820
[2200] https://github.com/STEllAR-GROUP/hpx/issues/4818
[2201] https://github.com/STEllAR-GROUP/hpx/issues/4816
[2202] https://github.com/STEllAR-GROUP/hpx/issues/4813
[2203] https://github.com/STEllAR-GROUP/hpx/issues/4811
[2204] https://github.com/STEllAR-GROUP/hpx/issues/4810

- Issue #4787[2205] - *transform_inclusive_scan* gives incorrect results for non-commutative operator
- Issue #4786[2206] - transform_inclusive_scan tries to implicitly convert between types, instead of using the provided *conv* function
- Issue #4779[2207] - HPX build error with GCC 10.1
- Issue #4766[2208] - Move HPX.Compute functionality to experimental namespace
- Issue #4763[2209] - License file name
- Issue #4758[2210] - CMake profiling results
- Issue #4755[2211] - Building HPX with support for PAPI fails
- Issue #4754[2212] - CMake cache creation breaks when using HPX with mimalloc
- Issue #4752[2213] - HPX MPI Future build failed
- Issue #4746[2214] - Memory leak when using dataflow icw components
- Issue #4731[2215] - Bug in stencil example, calculation of locality IDs
- Issue #4723[2216] - Build fail with NETWORKING OFF
- Issue #4720[2217] - Add compatibility headers for modules that had their module headers implicitly generated in 1.4.1
- Issue #4719[2218] - Undeprecate some module headers
- Issue #4712[2219] - Rename HPX_MPI_WITH_FUTURES option
- Issue #4709[2220] - Make deprecation warnings overridable in dependent projects
- Issue #4691[2221] - Suggestion to fix and enhance the thread_mapper API
- Issue #4686[2222] - Fix tutorials examples
- Issue #4685[2223] - HPX distributed map fails to compile
- Issue #4680[2224] - Build error with HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- Issue #4679[2225] - Build error for hpx w/ Apex on Summit
- Issue #4675[2226] - build error with HPX_WITH_NETWORKING=OFF

2205 https://github.com/STEllAR-GROUP/hpx/issues/4787
2206 https://github.com/STEllAR-GROUP/hpx/issues/4786
2207 https://github.com/STEllAR-GROUP/hpx/issues/4779
2208 https://github.com/STEllAR-GROUP/hpx/issues/4766
2209 https://github.com/STEllAR-GROUP/hpx/issues/4763
2210 https://github.com/STEllAR-GROUP/hpx/issues/4758
2211 https://github.com/STEllAR-GROUP/hpx/issues/4755
2212 https://github.com/STEllAR-GROUP/hpx/issues/4754
2213 https://github.com/STEllAR-GROUP/hpx/issues/4752
2214 https://github.com/STEllAR-GROUP/hpx/issues/4746
2215 https://github.com/STEllAR-GROUP/hpx/issues/4731
2216 https://github.com/STEllAR-GROUP/hpx/issues/4723
2217 https://github.com/STEllAR-GROUP/hpx/issues/4720
2218 https://github.com/STEllAR-GROUP/hpx/issues/4719
2219 https://github.com/STEllAR-GROUP/hpx/issues/4712
2220 https://github.com/STEllAR-GROUP/hpx/issues/4709
2221 https://github.com/STEllAR-GROUP/hpx/issues/4691
2222 https://github.com/STEllAR-GROUP/hpx/issues/4686
2223 https://github.com/STEllAR-GROUP/hpx/issues/4685
2224 https://github.com/STEllAR-GROUP/hpx/issues/4680
2225 https://github.com/STEllAR-GROUP/hpx/issues/4679
2226 https://github.com/STEllAR-GROUP/hpx/issues/4675

- Issue #4674[2227] - Error running Quickstart tests on OS X
- Issue #4662[2228] - MPI initialization broken when networking off
- Issue #4652[2229] - How to fix distributed action annotation
- Issue #4650[2230] - thread descriptions are broken…again
- Issue #4648[2231] - Thread stacksize not properly set
- Issue #4647[2232] - Rename generated collective headers in modules
- Issue #4639[2233] - Update deprecation warnings in compatibility headers to point to collective headers
- Issue #4628[2234] - mpi parcelport totally broken
- Issue #4619[2235] - Fully document hpx_wrap behaviour and targets
- Issue #4612[2236] - Compilation issue with HPX 1.4.1 and 1.4.0
- Issue #4594[2237] - Rename modules
- Issue #4578[2238] - Default value for HPX_WITH_THREAD_BACKTRACE_DEPTH
- Issue #4572[2239] - Thread manager should be given a runtime_configuration
- Issue #4571[2240] - Add high-level documentation to new modules
- Issue #4569[2241] - Annoying warning when compiling - pls suppress or fix it.
- Issue #4555[2242] - HPX_HAVE_THREAD_BACKTRACE_ON_SUSPENSION compilation error
- Issue #4543[2243] - Segfaults in Release builds using *sleep_for*
- Issue #4539[2244] - Compilation Error when HPX_MPI_WITH_FUTURES=ON
- Issue #4537[2245] - Linking issue with libhpx_initd.a
- Issue #4535[2246] - API for checking if pool with a given name exists
- Issue #4523[2247] - Build of PR #4311 (git tag 9955e8e) fails
- Issue #4519[2248] - Documentation problem
- Issue #4513[2249] - HPXConfig.cmake contains ill-formed paths when library paths use backslashes

[2227] https://github.com/STEllAR-GROUP/hpx/issues/4674
[2228] https://github.com/STEllAR-GROUP/hpx/issues/4662
[2229] https://github.com/STEllAR-GROUP/hpx/issues/4652
[2230] https://github.com/STEllAR-GROUP/hpx/issues/4650
[2231] https://github.com/STEllAR-GROUP/hpx/issues/4648
[2232] https://github.com/STEllAR-GROUP/hpx/issues/4647
[2233] https://github.com/STEllAR-GROUP/hpx/issues/4639
[2234] https://github.com/STEllAR-GROUP/hpx/issues/4628
[2235] https://github.com/STEllAR-GROUP/hpx/issues/4619
[2236] https://github.com/STEllAR-GROUP/hpx/issues/4612
[2237] https://github.com/STEllAR-GROUP/hpx/issues/4594
[2238] https://github.com/STEllAR-GROUP/hpx/issues/4578
[2239] https://github.com/STEllAR-GROUP/hpx/issues/4572
[2240] https://github.com/STEllAR-GROUP/hpx/issues/4571
[2241] https://github.com/STEllAR-GROUP/hpx/issues/4569
[2242] https://github.com/STEllAR-GROUP/hpx/issues/4555
[2243] https://github.com/STEllAR-GROUP/hpx/issues/4543
[2244] https://github.com/STEllAR-GROUP/hpx/issues/4539
[2245] https://github.com/STEllAR-GROUP/hpx/issues/4537
[2246] https://github.com/STEllAR-GROUP/hpx/issues/4535
[2247] https://github.com/STEllAR-GROUP/hpx/issues/4523
[2248] https://github.com/STEllAR-GROUP/hpx/issues/4519
[2249] https://github.com/STEllAR-GROUP/hpx/issues/4513

- Issue #4507[2250] - User-polling introduced by MPI futures module should be more generally usable
- Issue #4506[2251] - Make sure force_linking.hpp is not included in main module header
- Issue #4501[2252] - Fix compilation of PAPI tests
- Issue #4497[2253] - Add modules CI checks
- Issue #4489[2254] - Polymorphic executor
- Issue #4476[2255] - Use CMake targets defined by FindBoost
- Issue #4473[2256] - Add vcpkg installation instructions
- Issue #4470[2257] - Adapt hpx::future to C++20 co_await
- Issue #4468[2258] - Compile error on Raspberry Pi 4
- Issue #4466[2259] - Compile error on Windows, current stable:
- Issue #4453[2260] - Installing HPX on fedora with dnf is not adding cmake files
- Issue #4448[2261] - New std::variant serialization broken
- Issue #4438[2262] - Add performance counter flag is monotically increasing
- Issue #4436[2263] - Build problem: same code build and works with 1.4.0 but it doesn't with 1.4.1
- Issue #4429[2264] - Function descriptions not supported in distributed
- Issue #4423[2265] - –hpx:ini=hpx.lock_detection=0 has no effect
- Issue #4422[2266] - Add performance counter metadata
- Issue #4419[2267] - Weird behavior for –hpx:print-counter-interval with large numbers
- Issue #4401[2268] - Create module repository
- Issue #4400[2269] - Command line options conflict related to performance counters
- Issue #4349[2270] - *–hpx:use-process-mask* option throw an exception on OS X
- Issue #4345[2271] - Move gh-pages branch out of hpx repo
- Issue #4323[2272] - Const-correctness error in assignment operator of compute::vector

2250 https://github.com/STEllAR-GROUP/hpx/issues/4507
2251 https://github.com/STEllAR-GROUP/hpx/issues/4506
2252 https://github.com/STEllAR-GROUP/hpx/issues/4501
2253 https://github.com/STEllAR-GROUP/hpx/issues/4497
2254 https://github.com/STEllAR-GROUP/hpx/issues/4489
2255 https://github.com/STEllAR-GROUP/hpx/issues/4476
2256 https://github.com/STEllAR-GROUP/hpx/issues/4473
2257 https://github.com/STEllAR-GROUP/hpx/issues/4470
2258 https://github.com/STEllAR-GROUP/hpx/issues/4468
2259 https://github.com/STEllAR-GROUP/hpx/issues/4466
2260 https://github.com/STEllAR-GROUP/hpx/issues/4453
2261 https://github.com/STEllAR-GROUP/hpx/issues/4448
2262 https://github.com/STEllAR-GROUP/hpx/issues/4438
2263 https://github.com/STEllAR-GROUP/hpx/issues/4436
2264 https://github.com/STEllAR-GROUP/hpx/issues/4429
2265 https://github.com/STEllAR-GROUP/hpx/issues/4423
2266 https://github.com/STEllAR-GROUP/hpx/issues/4422
2267 https://github.com/STEllAR-GROUP/hpx/issues/4419
2268 https://github.com/STEllAR-GROUP/hpx/issues/4401
2269 https://github.com/STEllAR-GROUP/hpx/issues/4400
2270 https://github.com/STEllAR-GROUP/hpx/issues/4349
2271 https://github.com/STEllAR-GROUP/hpx/issues/4345
2272 https://github.com/STEllAR-GROUP/hpx/issues/4323

- Issue #4318[2273] - ASIO breaks with C++2a concepts
- Issue #4317[2274] - Application runs even if *–hpx:help* is specified
- Issue #4063[2275] - Document hpxcxx compiler wrapper
- Issue #3983[2276] - Implement the C++20 Synchronization Library
- Issue #3696[2277] - C++11 *constexpr* support is now required
- Issue #3623[2278] - Modular HPX branch and an alternative project layout
- Issue #2836[2279] - The worst-case time complexity of parallel::sort seems to be O(N^2).

Closed pull requests

- PR #4936[2280] - Minor documentation fixes part 2
- PR #4935[2281] - Add copyright and license to joss paper file
- PR #4934[2282] - Adding Semicolon in Documentation
- PR #4932[2283] - Fixing compiler warnings
- PR #4931[2284] - Small documentation formatting fixes
- PR #4930[2285] - Documentation Distributed HPX applications localvv with local_vv
- PR #4929[2286] - Add final version of the JOSS paper
- PR #4928[2287] - Add HPX_NODISCARD to enable_user_polling structs
- PR #4926[2288] - Rename distributed_executors module to executors_distributed
- PR #4925[2289] - Making transform_reduce conforming to C++20
- PR #4923[2290] - Don't acquire lock if not needed
- PR #4921[2291] - Update the release notes for the release candidate 3
- PR #4920[2292] - Disable libcds release
- PR #4919[2293] - Make cuda event pool dynamic instead of fixed size
- PR #4917[2294] - Move chrono functionality to hpx::chrono namespace

[2273] https://github.com/STEllAR-GROUP/hpx/issues/4318
[2274] https://github.com/STEllAR-GROUP/hpx/issues/4317
[2275] https://github.com/STEllAR-GROUP/hpx/issues/4063
[2276] https://github.com/STEllAR-GROUP/hpx/issues/3983
[2277] https://github.com/STEllAR-GROUP/hpx/issues/3696
[2278] https://github.com/STEllAR-GROUP/hpx/issues/3623
[2279] https://github.com/STEllAR-GROUP/hpx/issues/2836
[2280] https://github.com/STEllAR-GROUP/hpx/pull/4936
[2281] https://github.com/STEllAR-GROUP/hpx/pull/4935
[2282] https://github.com/STEllAR-GROUP/hpx/pull/4934
[2283] https://github.com/STEllAR-GROUP/hpx/pull/4932
[2284] https://github.com/STEllAR-GROUP/hpx/pull/4931
[2285] https://github.com/STEllAR-GROUP/hpx/pull/4930
[2286] https://github.com/STEllAR-GROUP/hpx/pull/4929
[2287] https://github.com/STEllAR-GROUP/hpx/pull/4928
[2288] https://github.com/STEllAR-GROUP/hpx/pull/4926
[2289] https://github.com/STEllAR-GROUP/hpx/pull/4925
[2290] https://github.com/STEllAR-GROUP/hpx/pull/4923
[2291] https://github.com/STEllAR-GROUP/hpx/pull/4921
[2292] https://github.com/STEllAR-GROUP/hpx/pull/4920
[2293] https://github.com/STEllAR-GROUP/hpx/pull/4919
[2294] https://github.com/STEllAR-GROUP/hpx/pull/4917

- PR #4916[2295] - HPX_HAVE_DEPRECATION_WARNINGS needs to be set even when disabled
- PR #4915[2296] - Moving more action related files to actions modules
- PR #4914[2297] - Add alias targets with namespaces used for exporting
- PR #4912[2298] - Aggregate initialize CPOs
- PR #4910[2299] - Explicitly specify hwloc root on Jenkins CSCS builds
- PR #4908[2300] - Fix algorithms documentation
- PR #4907[2301] - Remove HPX::hpx_no_wrap_main target
- PR #4906[2302] - Fixing unused variable warning
- PR #4905[2303] - Adding specializations for simple for_loops
- PR #4904[2304] - Update boost to 1.74.0 for the newest jenkins configs
- PR #4903[2305] - Hide GITHUB_TOKEN environment variables from environment variable output
- PR #4902[2306] - Cancel previous pull requests builds before starting a new one with Jenkins
- PR #4901[2307] - Update public API list with updated algorithms
- PR #4899[2308] - Suggested changes for HPX V1.5 release notes
- PR #4898[2309] - Minor tweak to hpx::equal implementation
- PR #4896[2310] - Making generate() and generate_n conforming to C++20
- PR #4895[2311] - Update apex tag
- PR #4894[2312] - Fix exception handling for tasks
- PR #4893[2313] - Remove last use of std::result_of, removed in C++20
- PR #4892[2314] - Adding replay_executor and replicate_executor
- PR #4889[2315] - Restore old behaviour of not requiring linking to hpx_wrap when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4887[2316] - Making sure remotely thrown (non-hpx) exceptions are properly marshaled back to invocation site

2295 https://github.com/STEllAR-GROUP/hpx/pull/4916
2296 https://github.com/STEllAR-GROUP/hpx/pull/4915
2297 https://github.com/STEllAR-GROUP/hpx/pull/4914
2298 https://github.com/STEllAR-GROUP/hpx/pull/4912
2299 https://github.com/STEllAR-GROUP/hpx/pull/4910
2300 https://github.com/STEllAR-GROUP/hpx/pull/4908
2301 https://github.com/STEllAR-GROUP/hpx/pull/4907
2302 https://github.com/STEllAR-GROUP/hpx/pull/4906
2303 https://github.com/STEllAR-GROUP/hpx/pull/4905
2304 https://github.com/STEllAR-GROUP/hpx/pull/4904
2305 https://github.com/STEllAR-GROUP/hpx/pull/4903
2306 https://github.com/STEllAR-GROUP/hpx/pull/4902
2307 https://github.com/STEllAR-GROUP/hpx/pull/4901
2308 https://github.com/STEllAR-GROUP/hpx/pull/4899
2309 https://github.com/STEllAR-GROUP/hpx/pull/4898
2310 https://github.com/STEllAR-GROUP/hpx/pull/4896
2311 https://github.com/STEllAR-GROUP/hpx/pull/4895
2312 https://github.com/STEllAR-GROUP/hpx/pull/4894
2313 https://github.com/STEllAR-GROUP/hpx/pull/4893
2314 https://github.com/STEllAR-GROUP/hpx/pull/4892
2315 https://github.com/STEllAR-GROUP/hpx/pull/4889
2316 https://github.com/STEllAR-GROUP/hpx/pull/4887

- PR #4885[2317] - Adapting hpx::find and friends to C++20
- PR #4884[2318] - Adapting mismatch to C++20
- PR #4883[2319] - Adapting hpx::equal to be conforming to C++20
- PR #4882[2320] - Fixing exception handling for hpx::copy and adding missing tests
- PR #4881[2321] - Adds different runtime exception when registering thread with the HPX runtime
- PR #4876[2322] - Adding example demonstrating how to disable thread stealing during the execution of parallel algorithms
- PR #4874[2323] - Adding non-policy tests to all_of, any_of, and none_of
- PR #4873[2324] - Set CUDA compute capability on rostam Jenkins builds
- PR #4872[2325] - Force partitioned vector scan tests to run serially
- PR #4870[2326] - Making move conforming with C++20
- PR #4869[2327] - Making destroy and destroy_n conforming to C++20
- PR #4868[2328] - Fix miscellaneous header problems
- PR #4867[2329] - Add CPOs for for_each
- PR #4865[2330] - Adapting count and count_if to be conforming to C++20
- PR #4864[2331] - Release notes 1.5.0
- PR #4863[2332] - adding libcds-hpx tag to prepare for hpx1.5 release
- PR #4862[2333] - Adding version specific deprecation options
- PR #4861[2334] - Limiting executor improvements
- PR #4860[2335] - Making fill and fill_n compatible with C++20
- PR #4859[2336] - Adapting all_of, any_of, and none_of to C++20
- PR #4857[2337] - Improve libCDS integration
- PR #4856[2338] - Correct typos in the documentation of the hpx performance counters
- PR #4854[2339] - Removing obsolete code

[2317] https://github.com/STEllAR-GROUP/hpx/pull/4885
[2318] https://github.com/STEllAR-GROUP/hpx/pull/4884
[2319] https://github.com/STEllAR-GROUP/hpx/pull/4883
[2320] https://github.com/STEllAR-GROUP/hpx/pull/4882
[2321] https://github.com/STEllAR-GROUP/hpx/pull/4881
[2322] https://github.com/STEllAR-GROUP/hpx/pull/4876
[2323] https://github.com/STEllAR-GROUP/hpx/pull/4874
[2324] https://github.com/STEllAR-GROUP/hpx/pull/4873
[2325] https://github.com/STEllAR-GROUP/hpx/pull/4872
[2326] https://github.com/STEllAR-GROUP/hpx/pull/4870
[2327] https://github.com/STEllAR-GROUP/hpx/pull/4869
[2328] https://github.com/STEllAR-GROUP/hpx/pull/4868
[2329] https://github.com/STEllAR-GROUP/hpx/pull/4867
[2330] https://github.com/STEllAR-GROUP/hpx/pull/4865
[2331] https://github.com/STEllAR-GROUP/hpx/pull/4864
[2332] https://github.com/STEllAR-GROUP/hpx/pull/4863
[2333] https://github.com/STEllAR-GROUP/hpx/pull/4862
[2334] https://github.com/STEllAR-GROUP/hpx/pull/4861
[2335] https://github.com/STEllAR-GROUP/hpx/pull/4860
[2336] https://github.com/STEllAR-GROUP/hpx/pull/4859
[2337] https://github.com/STEllAR-GROUP/hpx/pull/4857
[2338] https://github.com/STEllAR-GROUP/hpx/pull/4856
[2339] https://github.com/STEllAR-GROUP/hpx/pull/4854

- PR #4853[2340] - Adding test that derives component from two other components
- PR #4852[2341] - Fix mpi_ring test in distributed mode by ensuring all ranks run hpx_main
- PR #4851[2342] - Converting resiliency APIs to tag_invoke based CPOs
- PR #4849[2343] - Enable use of future_overhead test when DISTRIBUTED_RUNTIME is OFF
- PR #4847[2344] - Fixing 'error prone' constructs as reported by Codacy
- PR #4846[2345] - Disable Boost.Asio concepts support
- PR #4845[2346] - Fix PAPI counters
- PR #4843[2347] - Remove dependency on various Boost headers
- PR #4841[2348] - Rearrange public API headers
- PR #4840[2349] - Fixing TSS problems during thread termination
- PR #4839[2350] - Fix async_cuda build problems when distributed runtime is disabled
- PR #4837[2351] - Restore compatibility for old (now deprecated) copy algorithms
- PR #4836[2352] - Adding CPOs for hpx::reduce
- PR #4835[2353] - Remove *using util::result_of* from namespace hpx
- PR #4834[2354] - Fixing the calculation of the number of idle cores and the corresponding idle masks
- PR #4833[2355] - Allow thread function destructors to yield
- PR #4832[2356] - Fixing assertion in split_gids and memory leaks in 1d_stencil_7
- PR #4831[2357] - Making sure MPI_CXX_COMPILE_FLAGS is interpreted as a sequence of options
- PR #4830[2358] - Update documentation on using HPX::wrap_main
- PR #4827[2359] - Update clang-newest configuration to use clang 10
- PR #4826[2360] - Add Jenkins configuration for rostam
- PR #4825[2361] - Move all CUDA functionality to hpx::cuda::experimental namespace
- PR #4824[2362] - Add support for building master/release branches to Jenkins configuration

[2340] https://github.com/STEllAR-GROUP/hpx/pull/4853
[2341] https://github.com/STEllAR-GROUP/hpx/pull/4852
[2342] https://github.com/STEllAR-GROUP/hpx/pull/4851
[2343] https://github.com/STEllAR-GROUP/hpx/pull/4849
[2344] https://github.com/STEllAR-GROUP/hpx/pull/4847
[2345] https://github.com/STEllAR-GROUP/hpx/pull/4846
[2346] https://github.com/STEllAR-GROUP/hpx/pull/4845
[2347] https://github.com/STEllAR-GROUP/hpx/pull/4843
[2348] https://github.com/STEllAR-GROUP/hpx/pull/4841
[2349] https://github.com/STEllAR-GROUP/hpx/pull/4840
[2350] https://github.com/STEllAR-GROUP/hpx/pull/4839
[2351] https://github.com/STEllAR-GROUP/hpx/pull/4837
[2352] https://github.com/STEllAR-GROUP/hpx/pull/4836
[2353] https://github.com/STEllAR-GROUP/hpx/pull/4835
[2354] https://github.com/STEllAR-GROUP/hpx/pull/4834
[2355] https://github.com/STEllAR-GROUP/hpx/pull/4833
[2356] https://github.com/STEllAR-GROUP/hpx/pull/4832
[2357] https://github.com/STEllAR-GROUP/hpx/pull/4831
[2358] https://github.com/STEllAR-GROUP/hpx/pull/4830
[2359] https://github.com/STEllAR-GROUP/hpx/pull/4827
[2360] https://github.com/STEllAR-GROUP/hpx/pull/4826
[2361] https://github.com/STEllAR-GROUP/hpx/pull/4825
[2362] https://github.com/STEllAR-GROUP/hpx/pull/4824

- PR #4821[2363] - Implement customization point for hpx::copy and hpx::ranges::copy
- PR #4819[2364] - Allow finding Boost components before finding HPX
- PR #4817[2365] - Adding range version of stable sort
- PR #4815[2366] - Fix a wrong #ifdef for IO/TIMER pools causing build errors
- PR #4814[2367] - Replace hpx::function_nonser with std::function in error module
- PR #4809[2368] - Foreach adapt
- PR #4808[2369] - Make internal algorithms functions const
- PR #4807[2370] - Add Jenkins configuration for running on Piz Daint
- PR #4806[2371] - Update documentation links to new domain name
- PR #4805[2372] - Applying changes that resolve time complexity issues in sort
- PR #4803[2373] - Adding implementation of stable_sort
- PR #4802[2374] - Fix datapar header paths
- PR #4801[2375] - Replace boost::shared_array<T> with std::shared_ptr<T[]> if supported
- PR #4799[2376] - Fixing #include paths in compatibility headers
- PR #4798[2377] - Include the main module header (fixes partially #4488)
- PR #4797[2378] - Change cmake targets
- PR #4794[2379] - Removing 128bit integer emulation
- PR #4793[2380] - Make sure global variable is handled properly
- PR #4792[2381] - Replace enable_if with **HPX_CONCEPT_REQUIRES_** and add is_sentinel_for constraint
- PR #4790[2382] - Move deprecation warnings from base template to template specializations for result_of etc. structs
- PR #4789[2383] - Fix hangs during assertion handling and distributed runtime construction
- PR #4788[2384] - Fixing inclusive transform scan algorithm to properly handle initial value
- PR #4785[2385] - Fixing barrier test

2363 https://github.com/STEllAR-GROUP/hpx/pull/4821
2364 https://github.com/STEllAR-GROUP/hpx/pull/4819
2365 https://github.com/STEllAR-GROUP/hpx/pull/4817
2366 https://github.com/STEllAR-GROUP/hpx/pull/4815
2367 https://github.com/STEllAR-GROUP/hpx/pull/4814
2368 https://github.com/STEllAR-GROUP/hpx/pull/4809
2369 https://github.com/STEllAR-GROUP/hpx/pull/4808
2370 https://github.com/STEllAR-GROUP/hpx/pull/4807
2371 https://github.com/STEllAR-GROUP/hpx/pull/4806
2372 https://github.com/STEllAR-GROUP/hpx/pull/4805
2373 https://github.com/STEllAR-GROUP/hpx/pull/4803
2374 https://github.com/STEllAR-GROUP/hpx/pull/4802
2375 https://github.com/STEllAR-GROUP/hpx/pull/4801
2376 https://github.com/STEllAR-GROUP/hpx/pull/4799
2377 https://github.com/STEllAR-GROUP/hpx/pull/4798
2378 https://github.com/STEllAR-GROUP/hpx/pull/4797
2379 https://github.com/STEllAR-GROUP/hpx/pull/4794
2380 https://github.com/STEllAR-GROUP/hpx/pull/4793
2381 https://github.com/STEllAR-GROUP/hpx/pull/4792
2382 https://github.com/STEllAR-GROUP/hpx/pull/4790
2383 https://github.com/STEllAR-GROUP/hpx/pull/4789
2384 https://github.com/STEllAR-GROUP/hpx/pull/4788
2385 https://github.com/STEllAR-GROUP/hpx/pull/4785

- PR #4784[2386] - Fixing deleter argument bindings in serialize_buffer
- PR #4783[2387] - Add coveralls badge
- PR #4782[2388] - Make header tests parallel again
- PR #4780[2389] - Remove outdated comment about hpx::stop in documentation
- PR #4776[2390] - debug print improvements
- PR #4775[2391] - Checkpoint cleanup
- PR #4771[2392] - Fix compilation with HPX_WITH_NETWORKING=OFF
- PR #4767[2393] - Remove all force linking leftovers
- PR #4765[2394] - Fix 1d stencil index calculation
- PR #4764[2395] - Force some tests to run serially
- PR #4762[2396] - Update pointees in compatibility headers
- PR #4761[2397] - Fix running and building of execution module tests on CircleCI
- PR #4760[2398] - Storing hpx_options in global property to speed up summary report
- PR #4759[2399] - Reduce memory requirements for our main shared state
- PR #4757[2400] - Fix mimalloc linking on Windows
- PR #4756[2401] - Fix compilation issues
- PR #4753[2402] - Re-adding API functions that were lost during merges
- PR #4751[2403] - Revert "Create coverage reports and upload them to codecov.io"
- PR #4750[2404] - Fixing possible race condition during termination detection
- PR #4749[2405] - Deprecate result_of and friends
- PR #4748[2406] - Create coverage reports and upload them to codecov.io
- PR #4747[2407] - Changing #include for MPI parcelport
- PR #4745[2408] - Add *is_sentinel_for* trait implementation and test

[2386] https://github.com/STEllAR-GROUP/hpx/pull/4784
[2387] https://github.com/STEllAR-GROUP/hpx/pull/4783
[2388] https://github.com/STEllAR-GROUP/hpx/pull/4782
[2389] https://github.com/STEllAR-GROUP/hpx/pull/4780
[2390] https://github.com/STEllAR-GROUP/hpx/pull/4776
[2391] https://github.com/STEllAR-GROUP/hpx/pull/4775
[2392] https://github.com/STEllAR-GROUP/hpx/pull/4771
[2393] https://github.com/STEllAR-GROUP/hpx/pull/4767
[2394] https://github.com/STEllAR-GROUP/hpx/pull/4765
[2395] https://github.com/STEllAR-GROUP/hpx/pull/4764
[2396] https://github.com/STEllAR-GROUP/hpx/pull/4762
[2397] https://github.com/STEllAR-GROUP/hpx/pull/4761
[2398] https://github.com/STEllAR-GROUP/hpx/pull/4760
[2399] https://github.com/STEllAR-GROUP/hpx/pull/4759
[2400] https://github.com/STEllAR-GROUP/hpx/pull/4757
[2401] https://github.com/STEllAR-GROUP/hpx/pull/4756
[2402] https://github.com/STEllAR-GROUP/hpx/pull/4753
[2403] https://github.com/STEllAR-GROUP/hpx/pull/4751
[2404] https://github.com/STEllAR-GROUP/hpx/pull/4750
[2405] https://github.com/STEllAR-GROUP/hpx/pull/4749
[2406] https://github.com/STEllAR-GROUP/hpx/pull/4748
[2407] https://github.com/STEllAR-GROUP/hpx/pull/4747
[2408] https://github.com/STEllAR-GROUP/hpx/pull/4745

- PR #4743[2409] - Fix init_globally example after runtime mode changes
- PR #4742[2410] - Update SUPPORT.md
- PR #4741[2411] - Fixing a warning generated for unity builds with msvc
- PR #4740[2412] - Rename local_lcos and basic_execution modules
- PR #4739[2413] - Undeprecate a couple of hpx/modulename.hpp headers
- PR #4738[2414] - Conditionally test schedulers in thread_stacksize_current test
- PR #4734[2415] - Fixing a bunch of codacy warnings
- PR #4733[2416] - Add experimental unity build option to CMake configuration
- PR #4730[2417] - Fixing compilation problems with unordered map
- PR #4729[2418] - Fix APEX build
- PR #4727[2419] - Fix missing runtime includes for distributed runtime
- PR #4726[2420] - Add more API headers
- PR #4725[2421] - Add more compatibility headers for deprecated module headers
- PR #4724[2422] - Fix 4723
- PR #4721[2423] - Attempt to fixing migration tests
- PR #4717[2424] - Make the compatilibility headers macro conditional
- PR #4716[2425] - Add hpx/runtime.hpp and hpx/distributed/runtime.hpp API headers
- PR #4714[2426] - Add hpx/future.hpp header
- PR #4713[2427] - Remove hpx/runtime/threads_fwd.hpp and hpx/util_fwd.hpp
- PR #4711[2428] - Make module deprecation warnings overridable
- PR #4710[2429] - Add compatibility headers and other fixes after module header renaming
- PR #4708[2430] - Add termination handler for parallel algorithms
- PR #4707[2431] - Use hpx::function_nonser instead of std::function internally

[2409] https://github.com/STEllAR-GROUP/hpx/pull/4743
[2410] https://github.com/STEllAR-GROUP/hpx/pull/4742
[2411] https://github.com/STEllAR-GROUP/hpx/pull/4741
[2412] https://github.com/STEllAR-GROUP/hpx/pull/4740
[2413] https://github.com/STEllAR-GROUP/hpx/pull/4739
[2414] https://github.com/STEllAR-GROUP/hpx/pull/4738
[2415] https://github.com/STEllAR-GROUP/hpx/pull/4734
[2416] https://github.com/STEllAR-GROUP/hpx/pull/4733
[2417] https://github.com/STEllAR-GROUP/hpx/pull/4730
[2418] https://github.com/STEllAR-GROUP/hpx/pull/4729
[2419] https://github.com/STEllAR-GROUP/hpx/pull/4727
[2420] https://github.com/STEllAR-GROUP/hpx/pull/4726
[2421] https://github.com/STEllAR-GROUP/hpx/pull/4725
[2422] https://github.com/STEllAR-GROUP/hpx/pull/4724
[2423] https://github.com/STEllAR-GROUP/hpx/pull/4721
[2424] https://github.com/STEllAR-GROUP/hpx/pull/4717
[2425] https://github.com/STEllAR-GROUP/hpx/pull/4716
[2426] https://github.com/STEllAR-GROUP/hpx/pull/4714
[2427] https://github.com/STEllAR-GROUP/hpx/pull/4713
[2428] https://github.com/STEllAR-GROUP/hpx/pull/4711
[2429] https://github.com/STEllAR-GROUP/hpx/pull/4710
[2430] https://github.com/STEllAR-GROUP/hpx/pull/4708
[2431] https://github.com/STEllAR-GROUP/hpx/pull/4707

- PR #4706[2432] - Move header file to module
- PR #4705[2433] - Fix incorrect behaviour of cmake-format check
- PR #4704[2434] - Fix resource tests
- PR #4701[2435] - Fix missing includes for future::then specializations
- PR #4700[2436] - Removing obsolete memory component
- PR #4699[2437] - Add short descriptions to modules missing documentation
- PR #4696[2438] - Rename generated modules headers
- PR #4693[2439] - Overhauling thread_mapper for public consumption
- PR #4688[2440] - Fix thread stack size handling
- PR #4687[2441] - Adding all_gather and fixing all_to_all
- PR #4684[2442] - Miscellaneous compilation fixes
- PR #4683[2443] - Fix HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4682[2444] - Fix compilation of pack_traversal_rebind_container.hpp
- PR #4681[2445] - Add missing hpx/execution.hpp includes for future::then
- PR #4678[2446] - Typeless communicator
- PR #4677[2447] - Forcing registry option to be accepted without checks.
- PR #4676[2448] - Adding scatter_to/scatter_from collective operations
- PR #4673[2449] - Fix PAPI counters compilation
- PR #4671[2450] - Deprecate hpx::promise alias to hpx::lcos::promise
- PR #4670[2451] - Explicitly instantiate get_exception
- PR #4667[2452] - Add *stopValue* in *Sentinel* struct instead of *Iterator*
- PR #4666[2453] - Add release build on Windows to GitHub actions
- PR #4664[2454] - Creating itt_notify module.

[2432] https://github.com/STEllAR-GROUP/hpx/pull/4706
[2433] https://github.com/STEllAR-GROUP/hpx/pull/4705
[2434] https://github.com/STEllAR-GROUP/hpx/pull/4704
[2435] https://github.com/STEllAR-GROUP/hpx/pull/4701
[2436] https://github.com/STEllAR-GROUP/hpx/pull/4700
[2437] https://github.com/STEllAR-GROUP/hpx/pull/4699
[2438] https://github.com/STEllAR-GROUP/hpx/pull/4696
[2439] https://github.com/STEllAR-GROUP/hpx/pull/4693
[2440] https://github.com/STEllAR-GROUP/hpx/pull/4688
[2441] https://github.com/STEllAR-GROUP/hpx/pull/4687
[2442] https://github.com/STEllAR-GROUP/hpx/pull/4684
[2443] https://github.com/STEllAR-GROUP/hpx/pull/4683
[2444] https://github.com/STEllAR-GROUP/hpx/pull/4682
[2445] https://github.com/STEllAR-GROUP/hpx/pull/4681
[2446] https://github.com/STEllAR-GROUP/hpx/pull/4678
[2447] https://github.com/STEllAR-GROUP/hpx/pull/4677
[2448] https://github.com/STEllAR-GROUP/hpx/pull/4676
[2449] https://github.com/STEllAR-GROUP/hpx/pull/4673
[2450] https://github.com/STEllAR-GROUP/hpx/pull/4671
[2451] https://github.com/STEllAR-GROUP/hpx/pull/4670
[2452] https://github.com/STEllAR-GROUP/hpx/pull/4667
[2453] https://github.com/STEllAR-GROUP/hpx/pull/4666
[2454] https://github.com/STEllAR-GROUP/hpx/pull/4664

- PR #4663[2455] - Mpi fixes
- PR #4659[2456] - Making sure declarations match definitions in register_locks implementation
- PR #4655[2457] - Fixing task annotations for actions
- PR #4653[2458] - Making sure APEX is linked into every application, if needed
- PR #4651[2459] - Update get_function_annotation.hpp
- PR #4646[2460] - Runtime type
- PR #4645[2461] - Add a few more API headers
- PR #4644[2462] - Fixing support for mpirun (and similar)
- PR #4643[2463] - Fixing the fix for get_idle_core_count() API
- PR #4638[2464] - Remove HPX_API_EXPORT missed in previous cleanup
- PR #4636[2465] - Adding C++20 barrier
- PR #4635[2466] - Adding C++20 latch API
- PR #4634[2467] - Adding C++20 counting semaphore API
- PR #4633[2468] - Unify execution parameters customization points
- PR #4632[2469] - Adding missing bulk_sync_execute wrapper to example executor
- PR #4631[2470] - Updates to documentation; grammar edits.
- PR #4630[2471] - Updates to documentation; moved hyperlink
- PR #4624[2472] - Export set_self_ptr in thread_data.hpp instead of with forward declarations where used
- PR #4623[2473] - Clean up export macros
- PR #4621[2474] - Trigger an error for older boost versions on power architectures
- PR #4617[2475] - Ignore user-set compatibility header options if the module does not have compatibility headers
- PR #4616[2476] - Fix cmake-format warning
- PR #4615[2477] - Add handler for serializing custom exceptions

[2455] https://github.com/STEllAR-GROUP/hpx/pull/4663
[2456] https://github.com/STEllAR-GROUP/hpx/pull/4659
[2457] https://github.com/STEllAR-GROUP/hpx/pull/4655
[2458] https://github.com/STEllAR-GROUP/hpx/pull/4653
[2459] https://github.com/STEllAR-GROUP/hpx/pull/4651
[2460] https://github.com/STEllAR-GROUP/hpx/pull/4646
[2461] https://github.com/STEllAR-GROUP/hpx/pull/4645
[2462] https://github.com/STEllAR-GROUP/hpx/pull/4644
[2463] https://github.com/STEllAR-GROUP/hpx/pull/4643
[2464] https://github.com/STEllAR-GROUP/hpx/pull/4638
[2465] https://github.com/STEllAR-GROUP/hpx/pull/4636
[2466] https://github.com/STEllAR-GROUP/hpx/pull/4635
[2467] https://github.com/STEllAR-GROUP/hpx/pull/4634
[2468] https://github.com/STEllAR-GROUP/hpx/pull/4633
[2469] https://github.com/STEllAR-GROUP/hpx/pull/4632
[2470] https://github.com/STEllAR-GROUP/hpx/pull/4631
[2471] https://github.com/STEllAR-GROUP/hpx/pull/4630
[2472] https://github.com/STEllAR-GROUP/hpx/pull/4624
[2473] https://github.com/STEllAR-GROUP/hpx/pull/4623
[2474] https://github.com/STEllAR-GROUP/hpx/pull/4621
[2475] https://github.com/STEllAR-GROUP/hpx/pull/4617
[2476] https://github.com/STEllAR-GROUP/hpx/pull/4616
[2477] https://github.com/STEllAR-GROUP/hpx/pull/4615

- PR #4614[2478] - Fix error message when HPX_IGNORE_CMAKE_BUILD_TYPE_COMPATIBILITY=OFF
- PR #4613[2479] - Make partitioner constructor private
- PR #4611[2480] - Making auto_chunk_size execute the given function using the given executor
- PR #4610[2481] - Making sure the thread-local lock registration data is moving to the core the suspended HPX thread is resumed on
- PR #4609[2482] - Adding an API function that exposes the number of idle cores
- PR #4608[2483] - Fixing moodycamel namespace
- PR #4607[2484] - Moving winsocket initialization to core library
- PR #4606[2485] - Local runtime module etc.
- PR #4604[2486] - Add config_registry module
- PR #4603[2487] - Deal with distributed modules in their respective CMakeLists.txt
- PR #4602[2488] - Small module fixes
- PR #4598[2489] - Making sure current_executor and service_executor functions are linked into the core library
- PR #4597[2490] - Adding broadcast_to/broadcast_from to collectives module
- PR #4596[2491] - Fix performance regression in block_executor
- PR #4595[2492] - Making sure main.cpp is built as a library if HPX_WITH_DYNAMIC_MAIN=OFF
- PR #4592[2493] - Futures module
- PR #4591[2494] - Adapting co_await support for C++20
- PR #4590[2495] - Adding missing exception test for for_loop()
- PR #4587[2496] - Move traits headers to hpx/modulename/traits directory
- PR #4586[2497] - Remove Travis CI config
- PR #4585[2498] - Update macOS test blacklist
- PR #4584[2499] - Attempting to fix missing symbols in stack trace
- PR #4583[2500] - Fixing bad static_cast

[2478] https://github.com/STEllAR-GROUP/hpx/pull/4614
[2479] https://github.com/STEllAR-GROUP/hpx/pull/4613
[2480] https://github.com/STEllAR-GROUP/hpx/pull/4611
[2481] https://github.com/STEllAR-GROUP/hpx/pull/4610
[2482] https://github.com/STEllAR-GROUP/hpx/pull/4609
[2483] https://github.com/STEllAR-GROUP/hpx/pull/4608
[2484] https://github.com/STEllAR-GROUP/hpx/pull/4607
[2485] https://github.com/STEllAR-GROUP/hpx/pull/4606
[2486] https://github.com/STEllAR-GROUP/hpx/pull/4604
[2487] https://github.com/STEllAR-GROUP/hpx/pull/4603
[2488] https://github.com/STEllAR-GROUP/hpx/pull/4602
[2489] https://github.com/STEllAR-GROUP/hpx/pull/4598
[2490] https://github.com/STEllAR-GROUP/hpx/pull/4597
[2491] https://github.com/STEllAR-GROUP/hpx/pull/4596
[2492] https://github.com/STEllAR-GROUP/hpx/pull/4595
[2493] https://github.com/STEllAR-GROUP/hpx/pull/4592
[2494] https://github.com/STEllAR-GROUP/hpx/pull/4591
[2495] https://github.com/STEllAR-GROUP/hpx/pull/4590
[2496] https://github.com/STEllAR-GROUP/hpx/pull/4587
[2497] https://github.com/STEllAR-GROUP/hpx/pull/4586
[2498] https://github.com/STEllAR-GROUP/hpx/pull/4585
[2499] https://github.com/STEllAR-GROUP/hpx/pull/4584
[2500] https://github.com/STEllAR-GROUP/hpx/pull/4583

- PR #4582[2501] - Changing download url for Windows prerequisites to circumvent bandwidth limitations
- PR #4581[2502] - Adding missing using placeholder::_X
- PR #4579[2503] - Move get_stack_size_name and related functions
- PR #4575[2504] - Excluding unconditional definition of class backtrace from global header
- PR #4574[2505] - Changing return type of hardware_concurrency() to unsigned int
- PR #4570[2506] - Move tests to modules
- PR #4564[2507] - Reshuffle internal targets and add HPX::hpx_no_wrap_main target
- PR #4563[2508] - fix CMake option typo
- PR #4562[2509] - Unregister lock earlier to avoid holding it while suspending
- PR #4561[2510] - Adding test macros supporting custom output stream
- PR #4560[2511] - Making sure hash_any::operator()() is linked into core library
- PR #4559[2512] - Fixing compilation if HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION=On
- PR #4557[2513] - Improve spinlock implementation to perform better in high-contention situations
- PR #4553[2514] - Fix a runtime_ptr problem at shutdown when apex is enabled
- PR #4552[2515] - Add configuration option for making exceptions less noisy
- PR #4551[2516] - Clean up thread creation parameters
- PR #4549[2517] - Test FetchContent build on GitHub actions
- PR #4548[2518] - Fix stack size
- PR #4545[2519] - Fix header tests
- PR #4544[2520] - Fix a typo in sanitizer build
- PR #4541[2521] - Add API to check if a thread pool exists
- PR #4540[2522] - Making sure MPI support is enabled if MPI futures are used but networking is disabled
- PR #4538[2523] - Move channel documentation examples to examples directory

[2501] https://github.com/STEllAR-GROUP/hpx/pull/4582
[2502] https://github.com/STEllAR-GROUP/hpx/pull/4581
[2503] https://github.com/STEllAR-GROUP/hpx/pull/4579
[2504] https://github.com/STEllAR-GROUP/hpx/pull/4575
[2505] https://github.com/STEllAR-GROUP/hpx/pull/4574
[2506] https://github.com/STEllAR-GROUP/hpx/pull/4570
[2507] https://github.com/STEllAR-GROUP/hpx/pull/4564
[2508] https://github.com/STEllAR-GROUP/hpx/pull/4563
[2509] https://github.com/STEllAR-GROUP/hpx/pull/4562
[2510] https://github.com/STEllAR-GROUP/hpx/pull/4561
[2511] https://github.com/STEllAR-GROUP/hpx/pull/4560
[2512] https://github.com/STEllAR-GROUP/hpx/pull/4559
[2513] https://github.com/STEllAR-GROUP/hpx/pull/4557
[2514] https://github.com/STEllAR-GROUP/hpx/pull/4553
[2515] https://github.com/STEllAR-GROUP/hpx/pull/4552
[2516] https://github.com/STEllAR-GROUP/hpx/pull/4551
[2517] https://github.com/STEllAR-GROUP/hpx/pull/4549
[2518] https://github.com/STEllAR-GROUP/hpx/pull/4548
[2519] https://github.com/STEllAR-GROUP/hpx/pull/4545
[2520] https://github.com/STEllAR-GROUP/hpx/pull/4544
[2521] https://github.com/STEllAR-GROUP/hpx/pull/4541
[2522] https://github.com/STEllAR-GROUP/hpx/pull/4540
[2523] https://github.com/STEllAR-GROUP/hpx/pull/4538

- PR #4536[2524] - Add generic allocator for execution policies
- PR #4534[2525] - Enable compatibility headers for thread_executors module
- PR #4532[2526] - Fixing broken url in README.rst
- PR #4531[2527] - Update scripts
- PR #4530[2528] - Make sure module API docs show up in correct order
- PR #4529[2529] - Adding missing template code to module creation script
- PR #4528[2530] - Make sure version module uses HPX's binary dir, not the parent's
- PR #4527[2531] - Creating actions_base and actions module
- PR #4526[2532] - Shared state for cv
- PR #4525[2533] - Changing sub-name sequencing for experimental namespace
- PR #4524[2534] - Add API guarantee notes to API reference documentation
- PR #4522[2535] - Enable and fix deprecation warnings in execution module
- PR #4521[2536] - Moves more miscellaneous files to modules
- PR #4520[2537] - Skip execution customization points when executor is known
- PR #4518[2538] - Module distributed lcos
- PR #4516[2539] - Fix various builds
- PR #4515[2540] - Replace backslashes by slashes in windows paths
- PR #4514[2541] - Adding polymorphic_executor
- PR #4512[2542] - Adding C++20 jthread and stop_token
- PR #4510[2543] - Attempt to fix APEX linking in external packages again
- PR #4508[2544] - Only test pull requests (not all branches) with GitHub actions
- PR #4505[2545] - Fix duplicate linking in tests (ODR violations)
- PR #4504[2546] - Fix C++ standard handling

[2524] https://github.com/STEllAR-GROUP/hpx/pull/4536
[2525] https://github.com/STEllAR-GROUP/hpx/pull/4534
[2526] https://github.com/STEllAR-GROUP/hpx/pull/4532
[2527] https://github.com/STEllAR-GROUP/hpx/pull/4531
[2528] https://github.com/STEllAR-GROUP/hpx/pull/4530
[2529] https://github.com/STEllAR-GROUP/hpx/pull/4529
[2530] https://github.com/STEllAR-GROUP/hpx/pull/4528
[2531] https://github.com/STEllAR-GROUP/hpx/pull/4527
[2532] https://github.com/STEllAR-GROUP/hpx/pull/4526
[2533] https://github.com/STEllAR-GROUP/hpx/pull/4525
[2534] https://github.com/STEllAR-GROUP/hpx/pull/4524
[2535] https://github.com/STEllAR-GROUP/hpx/pull/4522
[2536] https://github.com/STEllAR-GROUP/hpx/pull/4521
[2537] https://github.com/STEllAR-GROUP/hpx/pull/4520
[2538] https://github.com/STEllAR-GROUP/hpx/pull/4518
[2539] https://github.com/STEllAR-GROUP/hpx/pull/4516
[2540] https://github.com/STEllAR-GROUP/hpx/pull/4515
[2541] https://github.com/STEllAR-GROUP/hpx/pull/4514
[2542] https://github.com/STEllAR-GROUP/hpx/pull/4512
[2543] https://github.com/STEllAR-GROUP/hpx/pull/4510
[2544] https://github.com/STEllAR-GROUP/hpx/pull/4508
[2545] https://github.com/STEllAR-GROUP/hpx/pull/4505
[2546] https://github.com/STEllAR-GROUP/hpx/pull/4504

- PR #4503[2547] - Add CMakelists file check
- PR #4500[2548] - Fix .clang-format version requirement comment
- PR #4499[2549] - Attempting to fix hpx_init linking on macOS
- PR #4498[2550] - Fix compatibility of *pool_executor*
- PR #4496[2551] - Removing superfluous SPDX tags
- PR #4494[2552] - Module executors
- PR #4493[2553] - Pack traversal module
- PR #4492[2554] - Update copyright year in documentation
- PR #4491[2555] - Add missing current_executor header
- PR #4490[2556] - Update GitHub actions configs
- PR #4487[2557] - Properly dispatch exceptions thrown from hpx_main to be rethrown from hpx::init/hpx::stop
- PR #4486[2558] - Fixing an initialization order problem
- PR #4485[2559] - Move miscellaneous files to their rightful modules
- PR #4483[2560] - Clean up imported CMake target naming
- PR #4481[2561] - Add vcpkg installation instructions
- PR #4479[2562] - Add hints to allow to specify MIMALLOC_ROOT
- PR #4478[2563] - Async modules
- PR #4475[2564] - Fix rp init changes
- PR #4474[2565] - Use #pragma once in headers
- PR #4472[2566] - Add more descriptive error message when using x86 coroutines on non-x86 platforms
- PR #4467[2567] - Add mimalloc find cmake script
- PR #4465[2568] - Add thread_executors module
- PR #4464[2569] - Include module

2547 https://github.com/STEllAR-GROUP/hpx/pull/4503
2548 https://github.com/STEllAR-GROUP/hpx/pull/4500
2549 https://github.com/STEllAR-GROUP/hpx/pull/4499
2550 https://github.com/STEllAR-GROUP/hpx/pull/4498
2551 https://github.com/STEllAR-GROUP/hpx/pull/4496
2552 https://github.com/STEllAR-GROUP/hpx/pull/4494
2553 https://github.com/STEllAR-GROUP/hpx/pull/4493
2554 https://github.com/STEllAR-GROUP/hpx/pull/4492
2555 https://github.com/STEllAR-GROUP/hpx/pull/4491
2556 https://github.com/STEllAR-GROUP/hpx/pull/4490
2557 https://github.com/STEllAR-GROUP/hpx/pull/4487
2558 https://github.com/STEllAR-GROUP/hpx/pull/4486
2559 https://github.com/STEllAR-GROUP/hpx/pull/4485
2560 https://github.com/STEllAR-GROUP/hpx/pull/4483
2561 https://github.com/STEllAR-GROUP/hpx/pull/4481
2562 https://github.com/STEllAR-GROUP/hpx/pull/4479
2563 https://github.com/STEllAR-GROUP/hpx/pull/4478
2564 https://github.com/STEllAR-GROUP/hpx/pull/4475
2565 https://github.com/STEllAR-GROUP/hpx/pull/4474
2566 https://github.com/STEllAR-GROUP/hpx/pull/4472
2567 https://github.com/STEllAR-GROUP/hpx/pull/4467
2568 https://github.com/STEllAR-GROUP/hpx/pull/4465
2569 https://github.com/STEllAR-GROUP/hpx/pull/4464

- PR #4462[2570] - Merge hpx_init and hpx_wrap into one static library
- PR #4461[2571] - Making thread_data test more realistic
- PR #4460[2572] - Suppress MPI warnings in version.cpp
- PR #4459[2573] - Make sure pkgconfig applications link with hpx_init
- PR #4458[2574] - Added example demonstrating how to create and use a wrapping executor
- PR #4457[2575] - Fixing execution of thread exit functions
- PR #4456[2576] - Move backtrace files to debugging module
- PR #4455[2577] - Move deadlock_detection and maintain_queue_wait_times source files into schedulers module
- PR #4450[2578] - Fixing compilation with std::filesystem enabled
- PR #4449[2579] - Fixing build system to actually build variant test
- PR #4447[2580] - This fixes an obsolete #include
- PR #4446[2581] - Resume tasks where they were suspended
- PR #4444[2582] - Minor CUDA fixes
- PR #4443[2583] - Add missing tests to CircleCI config
- PR #4442[2584] - Adding a tag to all auto-generated files allowing for tools to visually distinguish those
- PR #4441[2585] - Adding performance counter type information
- PR #4440[2586] - Fixing MSVC build
- PR #4439[2587] - Link HPX::plugin and component privately in hpx_setup_target
- PR #4437[2588] - Adding a test that verifies the problem can be solved using a trait specialization
- PR #4434[2589] - Clean up Boost dependencies and copy string algorithms to new module
- PR #4433[2590] - Fixing compilation issues (!) if MPI parcelport is enabled
- PR #4431[2591] - Ignore warnings about name mangling changing
- PR #4430[2592] - Add performance_counters module

[2570] https://github.com/STEllAR-GROUP/hpx/pull/4462
[2571] https://github.com/STEllAR-GROUP/hpx/pull/4461
[2572] https://github.com/STEllAR-GROUP/hpx/pull/4460
[2573] https://github.com/STEllAR-GROUP/hpx/pull/4459
[2574] https://github.com/STEllAR-GROUP/hpx/pull/4458
[2575] https://github.com/STEllAR-GROUP/hpx/pull/4457
[2576] https://github.com/STEllAR-GROUP/hpx/pull/4456
[2577] https://github.com/STEllAR-GROUP/hpx/pull/4455
[2578] https://github.com/STEllAR-GROUP/hpx/pull/4450
[2579] https://github.com/STEllAR-GROUP/hpx/pull/4449
[2580] https://github.com/STEllAR-GROUP/hpx/pull/4447
[2581] https://github.com/STEllAR-GROUP/hpx/pull/4446
[2582] https://github.com/STEllAR-GROUP/hpx/pull/4444
[2583] https://github.com/STEllAR-GROUP/hpx/pull/4443
[2584] https://github.com/STEllAR-GROUP/hpx/pull/4442
[2585] https://github.com/STEllAR-GROUP/hpx/pull/4441
[2586] https://github.com/STEllAR-GROUP/hpx/pull/4440
[2587] https://github.com/STEllAR-GROUP/hpx/pull/4439
[2588] https://github.com/STEllAR-GROUP/hpx/pull/4437
[2589] https://github.com/STEllAR-GROUP/hpx/pull/4434
[2590] https://github.com/STEllAR-GROUP/hpx/pull/4433
[2591] https://github.com/STEllAR-GROUP/hpx/pull/4431
[2592] https://github.com/STEllAR-GROUP/hpx/pull/4430

- PR #4428[2593] - Don't add compatibility headers to module API reference
- PR #4426[2594] - Add currently failing tests on GitHub actions to blacklist
- PR #4425[2595] - Clean up and correct minimum required versions
- PR #4424[2596] - Making sure hpx.lock_detection=0 works as advertized
- PR #4421[2597] - Making sure interval time stops underlying timer thread on termination
- PR #4417[2598] - Adding serialization support for std::variant (if available) and std::tuple
- PR #4415[2599] - Partially reverting changes applied by PR 4373
- PR #4414[2600] - Added documentation for the compiler-wrapper script hpxcxx.in in creating_hpx_projects.rst
- PR #4413[2601] - Merging from V1.4.1 release
- PR #4412[2602] - Making sure to issue a warning if a file specified using –hpx:options-file is not found
- PR #4411[2603] - Make test specific to HPX_WITH_SHARED_PRIORITY_SCHEDULER
- PR #4407[2604] - Adding minimal MPI executor
- PR #4405[2605] - Fix cross pool injection test, use default scheduler as falback
- PR #4404[2606] - Fix a race condition and clean-up usage of scheduler mode
- PR #4399[2607] - Add more threading modules
- PR #4398[2608] - Add CODEOWNERS file
- PR #4395[2609] - Adding a parameter to auto_chunk_size allowing to control the amount of iterations to measure
- PR #4393[2610] - Use appropriate cache-line size defaults for different platforms
- PR #4391[2611] - Fixing use of allocator for C++20
- PR #4390[2612] - Making –hpx:help behavior consistent
- PR #4388[2613] - Change the resource partitioner initialization
- PR #4387[2614] - Fix roll_release.sh
- PR #4386[2615] - Add warning messages for using thread binding options on macOS

[2593] https://github.com/STEllAR-GROUP/hpx/pull/4428
[2594] https://github.com/STEllAR-GROUP/hpx/pull/4426
[2595] https://github.com/STEllAR-GROUP/hpx/pull/4425
[2596] https://github.com/STEllAR-GROUP/hpx/pull/4424
[2597] https://github.com/STEllAR-GROUP/hpx/pull/4421
[2598] https://github.com/STEllAR-GROUP/hpx/pull/4417
[2599] https://github.com/STEllAR-GROUP/hpx/pull/4415
[2600] https://github.com/STEllAR-GROUP/hpx/pull/4414
[2601] https://github.com/STEllAR-GROUP/hpx/pull/4413
[2602] https://github.com/STEllAR-GROUP/hpx/pull/4412
[2603] https://github.com/STEllAR-GROUP/hpx/pull/4411
[2604] https://github.com/STEllAR-GROUP/hpx/pull/4407
[2605] https://github.com/STEllAR-GROUP/hpx/pull/4405
[2606] https://github.com/STEllAR-GROUP/hpx/pull/4404
[2607] https://github.com/STEllAR-GROUP/hpx/pull/4399
[2608] https://github.com/STEllAR-GROUP/hpx/pull/4398
[2609] https://github.com/STEllAR-GROUP/hpx/pull/4395
[2610] https://github.com/STEllAR-GROUP/hpx/pull/4393
[2611] https://github.com/STEllAR-GROUP/hpx/pull/4391
[2612] https://github.com/STEllAR-GROUP/hpx/pull/4390
[2613] https://github.com/STEllAR-GROUP/hpx/pull/4388
[2614] https://github.com/STEllAR-GROUP/hpx/pull/4387
[2615] https://github.com/STEllAR-GROUP/hpx/pull/4386

- PR #4385[2616] - Cuda futures
- PR #4384[2617] - Make enabling dynamic hpx_main on non-Linux systems a configuration error
- PR #4383[2618] - Use configure_file for HPXCacheVariables.cmake
- PR #4382[2619] - Update spellchecking whitelist and fix more typos
- PR #4380[2620] - Add a helper function to get a future from a cuda stream
- PR #4379[2621] - Add Windows and macOS CI with GitHub actions
- PR #4378[2622] - Change C++ standard handling
- PR #4377[2623] - Remove Python scripts
- PR #4374[2624] - Adding overload for *hpx::init*/*hpx::start* for use with resource partitioner
- PR #4373[2625] - Adding test that verifies for 4369 to be fixed
- PR #4372[2626] - Another attempt at fixing the integral mismatch and conversion warnings
- PR #4370[2627] - Doc updates quick start
- PR #4368[2628] - Add a whitelist of words for weird spelling suggestions
- PR #4366[2629] - Suppress or fix clang-tidy-9 warnings
- PR #4365[2630] - Removing more Boost dependencies
- PR #4363[2631] - Update clang-format config file for version 9
- PR #4362[2632] - Fix indices typo
- PR #4361[2633] - Boost cleanup
- PR #4360[2634] - Move plugins
- PR #4358[2635] - Doc updates; generating documentation. Will likely need heavy editing.
- PR #4356[2636] - Remove some minor unused and unnecessary Boost includes
- PR #4355[2637] - Fix spellcheck step in CircleCI config
- PR #4354[2638] - Lightweight utility to hold a pack as members

[2616] https://github.com/STEllAR-GROUP/hpx/pull/4385
[2617] https://github.com/STEllAR-GROUP/hpx/pull/4384
[2618] https://github.com/STEllAR-GROUP/hpx/pull/4383
[2619] https://github.com/STEllAR-GROUP/hpx/pull/4382
[2620] https://github.com/STEllAR-GROUP/hpx/pull/4380
[2621] https://github.com/STEllAR-GROUP/hpx/pull/4379
[2622] https://github.com/STEllAR-GROUP/hpx/pull/4378
[2623] https://github.com/STEllAR-GROUP/hpx/pull/4377
[2624] https://github.com/STEllAR-GROUP/hpx/pull/4374
[2625] https://github.com/STEllAR-GROUP/hpx/pull/4373
[2626] https://github.com/STEllAR-GROUP/hpx/pull/4372
[2627] https://github.com/STEllAR-GROUP/hpx/pull/4370
[2628] https://github.com/STEllAR-GROUP/hpx/pull/4368
[2629] https://github.com/STEllAR-GROUP/hpx/pull/4366
[2630] https://github.com/STEllAR-GROUP/hpx/pull/4365
[2631] https://github.com/STEllAR-GROUP/hpx/pull/4363
[2632] https://github.com/STEllAR-GROUP/hpx/pull/4362
[2633] https://github.com/STEllAR-GROUP/hpx/pull/4361
[2634] https://github.com/STEllAR-GROUP/hpx/pull/4360
[2635] https://github.com/STEllAR-GROUP/hpx/pull/4358
[2636] https://github.com/STEllAR-GROUP/hpx/pull/4356
[2637] https://github.com/STEllAR-GROUP/hpx/pull/4355
[2638] https://github.com/STEllAR-GROUP/hpx/pull/4354

- PR #4352[2639] - Minor fixes to the C++ standard detection for MSVC
- PR #4351[2640] - Move generated documentation to hpx-docs repo
- PR #4347[2641] - Add cmake policy - CMP0074
- PR #4346[2642] - Remove file committed by mistake
- PR #4342[2643] - Remove HCC and SYCL options from CMakeLists.txt
- PR #4341[2644] - Fix launch process test with APEX enabled
- PR #4340[2645] - Testing Cirrus CI
- PR #4339[2646] - Post 1.4.0 updates
- PR #4338[2647] - Spelling corrections and CircleCI spell check
- PR #4333[2648] - Flatten bound callables
- PR #4332[2649] - This is a collection of mostly minor (cleanup) fixes
- PR #4331[2650] - This adds the missing tests for async_colocated and async_continue_colocated
- PR #4330[2651] - Remove HPX.Compute host default_executor
- PR #4328[2652] - Generate global header for basic_execution module
- PR #4327[2653] - Use INTERNAL_FLAGS option for all examples and components
- PR #4326[2654] - Usage of temporary allocator in assignment operator of compute::vector
- PR #4325[2655] - Use hpx::threads::get_cache_line_size in prefetching.hpp
- PR #4324[2656] - Enable compatibility headers option for execution module
- PR #4316[2657] - Add clang format indentppdirectives
- PR #4313[2658] - Introduce index_pack alias to pack of size_t
- PR #4312[2659] - Fixing compatibility header for pack.hpp
- PR #4311[2660] - Dataflow annotations for APEX
- PR #4309[2661] - Update launching_and_configuring_hpx_applications.rst

[2639] https://github.com/STEllAR-GROUP/hpx/pull/4352
[2640] https://github.com/STEllAR-GROUP/hpx/pull/4351
[2641] https://github.com/STEllAR-GROUP/hpx/pull/4347
[2642] https://github.com/STEllAR-GROUP/hpx/pull/4346
[2643] https://github.com/STEllAR-GROUP/hpx/pull/4342
[2644] https://github.com/STEllAR-GROUP/hpx/pull/4341
[2645] https://github.com/STEllAR-GROUP/hpx/pull/4340
[2646] https://github.com/STEllAR-GROUP/hpx/pull/4339
[2647] https://github.com/STEllAR-GROUP/hpx/pull/4338
[2648] https://github.com/STEllAR-GROUP/hpx/pull/4333
[2649] https://github.com/STEllAR-GROUP/hpx/pull/4332
[2650] https://github.com/STEllAR-GROUP/hpx/pull/4331
[2651] https://github.com/STEllAR-GROUP/hpx/pull/4330
[2652] https://github.com/STEllAR-GROUP/hpx/pull/4328
[2653] https://github.com/STEllAR-GROUP/hpx/pull/4327
[2654] https://github.com/STEllAR-GROUP/hpx/pull/4326
[2655] https://github.com/STEllAR-GROUP/hpx/pull/4325
[2656] https://github.com/STEllAR-GROUP/hpx/pull/4324
[2657] https://github.com/STEllAR-GROUP/hpx/pull/4316
[2658] https://github.com/STEllAR-GROUP/hpx/pull/4313
[2659] https://github.com/STEllAR-GROUP/hpx/pull/4312
[2660] https://github.com/STEllAR-GROUP/hpx/pull/4311
[2661] https://github.com/STEllAR-GROUP/hpx/pull/4309

- PR #4306[2662] - Fix schedule hint not being taken from executor
- PR #4305[2663] - Implementing *hpx::functional::tag_invoke*
- PR #4304[2664] - Improve pack support utilities
- PR #4303[2665] - Remove errors module dependency on datastructures
- PR #4301[2666] - Clean up thread executors
- PR #4294[2667] - Logging revamp
- PR #4292[2668] - Remove SPDX tag from Boost License file to allow for github to recognize it
- PR #4291[2669] - Add format support for std::tm
- PR #4290[2670] - Simplify compatible tuples check
- PR #4288[2671] - A lightweight take on boost::lexical_cast
- PR #4287[2672] - Forking boost::lexical_cast as a new module
- PR #4277[2673] - MPI_futures
- PR #4270[2674] - Refactor future implementation
- PR #4265[2675] - Threading module
- PR #4259[2676] - Module naming base
- PR #4251[2677] - Local workrequesting scheduler
- PR #4250[2678] - Inline execution of scoped tasks, if possible
- PR #4247[2679] - Add execution in module headers
- PR #4246[2680] - Expose CMake targets officially
- PR #4239[2681] - Doc updates miscellaneous (partially completed during Google Season of Docs)
- PR #4233[2682] - Remove project() from modules + fix CMAKE_SOURCE_DIR issue
- PR #4231[2683] - Module local lcos
- PR #4207[2684] - Command line handling module

2662 https://github.com/STEllAR-GROUP/hpx/pull/4306
2663 https://github.com/STEllAR-GROUP/hpx/pull/4305
2664 https://github.com/STEllAR-GROUP/hpx/pull/4304
2665 https://github.com/STEllAR-GROUP/hpx/pull/4303
2666 https://github.com/STEllAR-GROUP/hpx/pull/4301
2667 https://github.com/STEllAR-GROUP/hpx/pull/4294
2668 https://github.com/STEllAR-GROUP/hpx/pull/4292
2669 https://github.com/STEllAR-GROUP/hpx/pull/4291
2670 https://github.com/STEllAR-GROUP/hpx/pull/4290
2671 https://github.com/STEllAR-GROUP/hpx/pull/4288
2672 https://github.com/STEllAR-GROUP/hpx/pull/4287
2673 https://github.com/STEllAR-GROUP/hpx/pull/4277
2674 https://github.com/STEllAR-GROUP/hpx/pull/4270
2675 https://github.com/STEllAR-GROUP/hpx/pull/4265
2676 https://github.com/STEllAR-GROUP/hpx/pull/4259
2677 https://github.com/STEllAR-GROUP/hpx/pull/4251
2678 https://github.com/STEllAR-GROUP/hpx/pull/4250
2679 https://github.com/STEllAR-GROUP/hpx/pull/4247
2680 https://github.com/STEllAR-GROUP/hpx/pull/4246
2681 https://github.com/STEllAR-GROUP/hpx/pull/4239
2682 https://github.com/STEllAR-GROUP/hpx/pull/4233
2683 https://github.com/STEllAR-GROUP/hpx/pull/4231
2684 https://github.com/STEllAR-GROUP/hpx/pull/4207

- PR #4206[2685] - Runtime configuration module
- PR #4141[2686] - Doc updates examples local to remote (partially completed during Google Season of Docs)
- PR #4091[2687] - Split runtime into local and distributed parts
- PR #4017[2688] - Require C++14

HPX V1.4.1 (Feb 12, 2020)

General changes

This is a bugfix release. It contains the following changes:

- Fix compilation issues on Windows, macOS, FreeBSD, and with gcc 10
- Install missing `pdb` files on Windows
- Allow running tests using an installed version of *HPX*
- Skip MPI finalization if HPX has not initialized MPI
- Give a hard error when attempting to use IO counters on Windows

Closed issues

- Issue #4320[2689] - HPX 1.4.0 does not compile with gcc 10
- Issue #4336[2690] - Building HPX 1.4.0 with IO Counters breaks (Windows)
- Issue #4334[2691] - HPX `Debug` and `RelWithDebinfo` builds on Windows not installing `.pdb` files
- Issue #4322[2692] - Undefine VT1 and VT2 after boost includes
- Issue #4314[2693] - Compile error on 1.4.0
- Issue #4307[2694] - `ld: error: duplicate symbol: freebsd_environ`

Closed pull requests

- PR #4376[2695] - Attempt to fix some test build errors on Windows
- PR #4357[2696] - Adding missing `#include`s to fix gcc V10 linker problems
- PR #4353[2697] - Skip `MPI_Finalize` if `MPI_Init` is not called from HPX
- PR #4343[2698] - Give a hard error if IO counters are enabled on non-Linux systems

[2685] https://github.com/STEllAR-GROUP/hpx/pull/4206
[2686] https://github.com/STEllAR-GROUP/hpx/pull/4141
[2687] https://github.com/STEllAR-GROUP/hpx/pull/4091
[2688] https://github.com/STEllAR-GROUP/hpx/pull/4017
[2689] https://github.com/STEllAR-GROUP/hpx/issues/4320
[2690] https://github.com/STEllAR-GROUP/hpx/issues/4336
[2691] https://github.com/STEllAR-GROUP/hpx/issues/4334
[2692] https://github.com/STEllAR-GROUP/hpx/issues/4322
[2693] https://github.com/STEllAR-GROUP/hpx/issues/4314
[2694] https://github.com/STEllAR-GROUP/hpx/issues/4307
[2695] https://github.com/STEllAR-GROUP/hpx/pull/4376
[2696] https://github.com/STEllAR-GROUP/hpx/pull/4357
[2697] https://github.com/STEllAR-GROUP/hpx/pull/4353
[2698] https://github.com/STEllAR-GROUP/hpx/pull/4343

- PR #4337[2699] - Installing pdb files on Windows
- PR #4335[2700] - Adding capability to buildsystem to use an installed version of HPX
- PR #4315[2701] - Forcing exported symbols from composable_guard to be linked into core library
- PR #4310[2702] - Remove environment handling from `exception.cpp`

HPX V1.4.0 (January 15, 2020)

General changes

- We have added the collectives `all_to_all` and `all_reduce`.
- We have added APIs for resiliency, which allows replication and replay for failed tasks. See the *documentation* for more details.
- Components can now be checkpointed.
- Performance improvements to schedulers and coroutines. A significant change is the addition of stackless coroutines. These are to be used for tasks that do not need to be suspended and can reduce overheads noticeably in applications with short tasks. A stackless coroutine can be created with the new stack size `thread_stacksize_nostack`.
- We have added an implementation of `unique_any`, which is a non-copyable version of `any`.
- The `shared_priority_queue_scheduler` has been improved. It now has lower overheads than the default scheduler in many situations. Unlike the default scheduler it fully supports NUMA scheduling hints. Enable it with the command line option `--hpx:queuing=shared-priority`. This scheduler should still be considered experimental, but its use is encouraged in real applications to help us make it production ready.
- We have added the performance counters `background-receive-duration` and `background-receive-overhead` for inspecting the time and overhead spent on receiving parcels in the background.
- Compilation time has been further improved when `HPX_WITH_NETWORKING=OFF`.
- We no longer require compiled Boost dependencies in certain configurations. This requires at least Boost 1.70, compiling on x86 with GCC 9, clang (libc++) 9, or VS2019 in C++17 mode. The dependency on Boost.Filesystem can explicitly be turned on with `HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY=ON` (it is off by default if the standard library supports `std::filesystem`). Boost.ProgramOptions has been copied into the HPX repository. We have a compatibility layer for users who must explicitly use Boost.ProgramOptions instead of the ProgramOptions provided by HPX. To remove the dependency `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY` must be explicitly set to `OFF`. This option will be removed in a future release. We have also removed several other header-only dependencies on Boost.
- It is now possible to use the process affinity mask set by tools like `numactl` and various batch environments with the command line option `--hpx:use-process-mask`. Enabling this option implies `--hpx:ignore-batch-env`.
- It is now possible to create standalone thread pools without starting the runtime. See the `standalone_thread_pool_executor.cpp` test in the `execution` module for an example.

[2699] https://github.com/STEllAR-GROUP/hpx/pull/4337
[2700] https://github.com/STEllAR-GROUP/hpx/pull/4335
[2701] https://github.com/STEllAR-GROUP/hpx/pull/4315
[2702] https://github.com/STEllAR-GROUP/hpx/pull/4310

- Tasks annotated with *hpx::util::annotated_function* now have their correct name when using APEX to generate OTF2 files.
- Cloning of APEX was defective in previous releases (it required manual intervention to check out the correct tag or branch). This has been fixed.
- The option `HPX_WITH_MORE_THAN_64_THREADS` is now ignored and will be removed in a future release. The value is instead derived directly from `HPX_WITH_MAX_CPU_COUNT` option.
- We have deprecated compiling in C++11 mode. The next release will require a C++14 capable compiler.
- We have deprecated support for the Vc library. This option will be replaced with SIMD support from the standard library in a future release.
- We have significantly refactored our CMake setup. This is intended to be a non-breaking change and will allow for using HPX through CMake targets in the future.
- We have continued modularizing the HPX library. In the process we have rearranged many header files into module-specific directories. All moved headers have compatibility headers which forward from the old location to the new location, together with a deprecation warning. The compatibility headers will eventually be removed.
- We now enforce formatting with `clang-format` on the majority of our source files.
- We have added SPDX license tags to all files.
- Many bugfixes.

Breaking changes

- The `HPX_WITH_THREAD_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_UNWRAPPED_COMPATIBLITY` option and the associated compatibility layer has been removed.

Closed issues

- Issue #4282[2703] - Build Issues with Release on Windows
- Issue #4278[2704] - Build Issues with CMake 3.14.4
- Issue #4273[2705] - Clients of HPX 1.4.0-rc2 with APEX ar not linked to libhpx-apex
- Issue #4269[2706] - Building HPX 1.4.0-rc2 with support for APEX fails
- Issue #4263[2707] - Compilation fail on latest master
- Issue #4232[2708] - Configure of HPX project using CMake FetchContent fails
- Issue #4223[2709] - "Re-using the main() function as the main HPX entry point" doesn't work
- Issue #4220[2710] - HPX won't compile - error building `resource_partitioner`

[2703] https://github.com/STEllAR-GROUP/hpx/issues/4282
[2704] https://github.com/STEllAR-GROUP/hpx/issues/4278
[2705] https://github.com/STEllAR-GROUP/hpx/issues/4273
[2706] https://github.com/STEllAR-GROUP/hpx/issues/4269
[2707] https://github.com/STEllAR-GROUP/hpx/issues/4263
[2708] https://github.com/STEllAR-GROUP/hpx/issues/4232
[2709] https://github.com/STEllAR-GROUP/hpx/issues/4223
[2710] https://github.com/STEllAR-GROUP/hpx/issues/4220

- Issue #4215[2711] - HPX 1.4.0rc1 does not link on s390x
- Issue #4204[2712] - Trouble compiling HPX with Intel compiler
- Issue #4199[2713] - Refactor APEX to eliminate circular dependency
- Issue #4187[2714] - HPX can't build on OSX
- Issue #4185[2715] - Simple debug output for development
- Issue #4182[2716] - `@HPX_CONF_PREFIX@` is the empty string
- Issue #4169[2717] - HPX won't build with APEX
- Issue #4163[2718] - Add back `HPX_LIBRARIES` and `HPX_INCLUDE_DIRS`
- Issue #4161[2719] - It should be possible to call `find_package(HPX)` multiple times
- Issue #4155[2720] - `get_self_id()` for stackless threads returns `invalid_thread_id`
- Issue #4151[2721] - build error with MPI code
- Issue #4150[2722] - hpx won't build on POWER9 with clang 8
- Issue #4148[2723] - `cacheline_data` delivers poor performance with C++17 compared to C++14
- Issue #4144[2724] - target general in `HPX_LIBRARIES` does not exist
- Issue #4134[2725] - CMake Error when `-DHPX_WITH_HPXMP=ON`
- Issue #4132[2726] - parallel fill leaves elements unfilled
- Issue #4123[2727] - PAPI performance counters are inaccessible
- Issue #4118[2728] - `static_chunk_size` is not obeyed in scan algorithms
- Issue #4115[2729] - dependency chaining error with APEX
- Issue #4107[2730] - Initializing runtime without entry point function and command line arguments
- Issue #4105[2731] - Bug in `hpx:bind=numa-balanced`
- Issue #4101[2732] - Bound tasks
- Issue #4100[2733] - Add SPDX identifier to all files

[2711] https://github.com/STEllAR-GROUP/hpx/issues/4215
[2712] https://github.com/STEllAR-GROUP/hpx/issues/4204
[2713] https://github.com/STEllAR-GROUP/hpx/issues/4199
[2714] https://github.com/STEllAR-GROUP/hpx/issues/4187
[2715] https://github.com/STEllAR-GROUP/hpx/issues/4185
[2716] https://github.com/STEllAR-GROUP/hpx/issues/4182
[2717] https://github.com/STEllAR-GROUP/hpx/issues/4169
[2718] https://github.com/STEllAR-GROUP/hpx/issues/4163
[2719] https://github.com/STEllAR-GROUP/hpx/issues/4161
[2720] https://github.com/STEllAR-GROUP/hpx/issues/4155
[2721] https://github.com/STEllAR-GROUP/hpx/issues/4151
[2722] https://github.com/STEllAR-GROUP/hpx/issues/4150
[2723] https://github.com/STEllAR-GROUP/hpx/issues/4148
[2724] https://github.com/STEllAR-GROUP/hpx/issues/4144
[2725] https://github.com/STEllAR-GROUP/hpx/issues/4134
[2726] https://github.com/STEllAR-GROUP/hpx/issues/4132
[2727] https://github.com/STEllAR-GROUP/hpx/issues/4123
[2728] https://github.com/STEllAR-GROUP/hpx/issues/4118
[2729] https://github.com/STEllAR-GROUP/hpx/issues/4115
[2730] https://github.com/STEllAR-GROUP/hpx/issues/4107
[2731] https://github.com/STEllAR-GROUP/hpx/issues/4105
[2732] https://github.com/STEllAR-GROUP/hpx/issues/4101
[2733] https://github.com/STEllAR-GROUP/hpx/issues/4100

- Issue #4085[2734] - `hpx_topology` library should depend on hwloc
- Issue #4067[2735] - HPX fails to build on macOS
- Issue #4056[2736] - Building without thread manager idle backoff fails
- Issue #4052[2737] - Enforce `clang-format` style for modules
- Issue #4032[2738] - Simple hello world fails to launch correctly
- Issue #4030[2739] - Allow threads to skip context switching
- Issue #4029[2740] - Add support for mimalloc
- Issue #4005[2741] - Can't link HPX when APEX enabled
- Issue #4002[2742] - Missing header for algorithm module
- Issue #3989[2743] - conversion from `long` to `unsigned int` requires a narrowing conversion on MSVC
- Issue #3958[2744] - `/statistics/average@` perf counter can't be created
- Issue #3953[2745] - CMake errors from `HPX_AddPseudoDependencies`
- Issue #3941[2746] - CMake error for APEX install target
- Issue #3940[2747] - Convert pseudo-doxygen function documentation into actual doxygen documentation
- Issue #3935[2748] - HPX compiler match too strict?
- Issue #3929[2749] - Buildbot failures on latest HPX stable
- Issue #3912[2750] - I recommend publishing a version that does not depend on the boost library
- Issue #3890[2751] - `hpx.ini` not working
- Issue #3883[2752] - cuda compilation fails because of `-faligned-new`
- Issue #3879[2753] - HPX fails to configure with `-DHPX_WITH_TESTS=OFF`
- Issue #3871[2754] - `dataflow` does not support void allocators
- Issue #3867[2755] - Latest HTML docs placed in wrong directory on GitHub pages
- Issue #3866[2756] - Make sure all tests use `HPX_TEST*` macros and not `HPX_ASSERT`

2734 https://github.com/STEllAR-GROUP/hpx/issues/4085
2735 https://github.com/STEllAR-GROUP/hpx/issues/4067
2736 https://github.com/STEllAR-GROUP/hpx/issues/4056
2737 https://github.com/STEllAR-GROUP/hpx/issues/4052
2738 https://github.com/STEllAR-GROUP/hpx/issues/4032
2739 https://github.com/STEllAR-GROUP/hpx/issues/4030
2740 https://github.com/STEllAR-GROUP/hpx/issues/4029
2741 https://github.com/STEllAR-GROUP/hpx/issues/4005
2742 https://github.com/STEllAR-GROUP/hpx/issues/4002
2743 https://github.com/STEllAR-GROUP/hpx/issues/3989
2744 https://github.com/STEllAR-GROUP/hpx/issues/3958
2745 https://github.com/STEllAR-GROUP/hpx/issues/3953
2746 https://github.com/STEllAR-GROUP/hpx/issues/3941
2747 https://github.com/STEllAR-GROUP/hpx/issues/3940
2748 https://github.com/STEllAR-GROUP/hpx/issues/3935
2749 https://github.com/STEllAR-GROUP/hpx/issues/3929
2750 https://github.com/STEllAR-GROUP/hpx/issues/3912
2751 https://github.com/STEllAR-GROUP/hpx/issues/3890
2752 https://github.com/STEllAR-GROUP/hpx/issues/3883
2753 https://github.com/STEllAR-GROUP/hpx/issues/3879
2754 https://github.com/STEllAR-GROUP/hpx/issues/3871
2755 https://github.com/STEllAR-GROUP/hpx/issues/3867
2756 https://github.com/STEllAR-GROUP/hpx/issues/3866

- Issue #3857[2757] - CMake all-keyword or all-plain for `target_link_libraries`
- Issue #3856[2758] - `hpx_setup_target` adds rogue flags
- Issue #3850[2759] - HPX fails to build on POWER8 with Clang7
- Issue #3848[2760] - Remove `lva` member from `thread_init_data`
- Issue #3838[2761] - `hpx::parallel::count/count_if` failing tests
- Issue #3651[2762] - `hpx::parallel::transform_reduce` with non const reference as lambda parameter
- Issue #3560[2763] - Apex integration with HPX not working properly
- Issue #3322[2764] - No warning when mixing debug/release builds

Closed pull requests

- PR #4300[2765] - Checks for `MPI_Init` being called twice
- PR #4299[2766] - Small CMake fixes
- PR #4298[2767] - Remove extra call to annotate function that messes up traces
- PR #4296[2768] - Fixing collectives locking problem
- PR #4295[2769] - Do not check `LICENSE_1_0.txt` for inspect violations
- PR #4293[2770] - Applying two small changes fixing carious MSVC/Windows problems
- PR #4285[2771] - Delete `apex.hpp`
- PR #4276[2772] - Disable doxygen generation for `hpx/debugging/print.hpp` file
- PR #4275[2773] - Make sure APEX is linked to even when not explicitly referenced
- PR #4272[2774] - Fix pushing of documentation
- PR #4271[2775] - Updating APEX tag, don't create new task_wrapper on `operator=` of hpx_thread object
- PR #4268[2776] - Testing for noexcept function specializations in C++11/14 mode
- PR #4267[2777] - Fixing MSVC warning
- PR #4266[2778] - Make sure macOS Travis CI fails if build step fails

2757 https://github.com/STEllAR-GROUP/hpx/issues/3857
2758 https://github.com/STEllAR-GROUP/hpx/issues/3856
2759 https://github.com/STEllAR-GROUP/hpx/issues/3850
2760 https://github.com/STEllAR-GROUP/hpx/issues/3848
2761 https://github.com/STEllAR-GROUP/hpx/issues/3838
2762 https://github.com/STEllAR-GROUP/hpx/issues/3651
2763 https://github.com/STEllAR-GROUP/hpx/issues/3560
2764 https://github.com/STEllAR-GROUP/hpx/issues/3322
2765 https://github.com/STEllAR-GROUP/hpx/pull/4300
2766 https://github.com/STEllAR-GROUP/hpx/pull/4299
2767 https://github.com/STEllAR-GROUP/hpx/pull/4298
2768 https://github.com/STEllAR-GROUP/hpx/pull/4296
2769 https://github.com/STEllAR-GROUP/hpx/pull/4295
2770 https://github.com/STEllAR-GROUP/hpx/pull/4293
2771 https://github.com/STEllAR-GROUP/hpx/pull/4285
2772 https://github.com/STEllAR-GROUP/hpx/pull/4276
2773 https://github.com/STEllAR-GROUP/hpx/pull/4275
2774 https://github.com/STEllAR-GROUP/hpx/pull/4272
2775 https://github.com/STEllAR-GROUP/hpx/pull/4271
2776 https://github.com/STEllAR-GROUP/hpx/pull/4268
2777 https://github.com/STEllAR-GROUP/hpx/pull/4267
2778 https://github.com/STEllAR-GROUP/hpx/pull/4266

- PR #4264[2779] - Clean up compatibility header options
- PR #4262[2780] - Cleanup modules `CMakeLists.txt`
- PR #4261[2781] - Fixing HPX/APEX linking and dependencies for external projects like Phylanx
- PR #4260[2782] - Fix docs compilation problems
- PR #4258[2783] - Couple of minor changes
- PR #4257[2784] - Fix apex annotation for async dispatch
- PR #4256[2785] - Remove lambdas from assert expressions
- PR #4255[2786] - Ignoring lock in `all_to_all` and `all_reduce`
- PR #4254[2787] - Adding action specializations for noexcept functions
- PR #4253[2788] - Move `partlit.hpp` to affinity module
- PR #4252[2789] - Make mismatching build types a hard error in CMake
- PR #4249[2790] - Scheduler improvement
- PR #4248[2791] - update hpxmp tag to v0.3.0
- PR #4245[2792] - Adding high performance channels
- PR #4244[2793] - Ignore lock in ignore_while_locked_1485 test
- PR #4243[2794] - Fix PAPI command line option documentation
- PR #4242[2795] - Ignore lock in target_distribution_policy
- PR #4241[2796] - Fix `start_stop_callbacks` test
- PR #4240[2797] - Mostly fix clang CUDA compilation
- PR #4238[2798] - Google Season of Docs updates to documentation; grammar edits.
- PR #4237[2799] - fixing annotated task to use the name, not the desc
- PR #4236[2800] - Move module print summary to modules
- PR #4235[2801] - Don't use alignas in `cache_{aligned,line}_data`

2779 https://github.com/STEllAR-GROUP/hpx/pull/4264
2780 https://github.com/STEllAR-GROUP/hpx/pull/4262
2781 https://github.com/STEllAR-GROUP/hpx/pull/4261
2782 https://github.com/STEllAR-GROUP/hpx/pull/4260
2783 https://github.com/STEllAR-GROUP/hpx/pull/4258
2784 https://github.com/STEllAR-GROUP/hpx/pull/4257
2785 https://github.com/STEllAR-GROUP/hpx/pull/4256
2786 https://github.com/STEllAR-GROUP/hpx/pull/4255
2787 https://github.com/STEllAR-GROUP/hpx/pull/4254
2788 https://github.com/STEllAR-GROUP/hpx/pull/4253
2789 https://github.com/STEllAR-GROUP/hpx/pull/4252
2790 https://github.com/STEllAR-GROUP/hpx/pull/4249
2791 https://github.com/STEllAR-GROUP/hpx/pull/4248
2792 https://github.com/STEllAR-GROUP/hpx/pull/4245
2793 https://github.com/STEllAR-GROUP/hpx/pull/4244
2794 https://github.com/STEllAR-GROUP/hpx/pull/4243
2795 https://github.com/STEllAR-GROUP/hpx/pull/4242
2796 https://github.com/STEllAR-GROUP/hpx/pull/4241
2797 https://github.com/STEllAR-GROUP/hpx/pull/4240
2798 https://github.com/STEllAR-GROUP/hpx/pull/4238
2799 https://github.com/STEllAR-GROUP/hpx/pull/4237
2800 https://github.com/STEllAR-GROUP/hpx/pull/4236
2801 https://github.com/STEllAR-GROUP/hpx/pull/4235

- PR #4234[2802] - Add basic overview sentence to all modules
- PR #4230[2803] - Add OS X builds to Travis CI
- PR #4229[2804] - Remove leftover queue compatibility checks
- PR #4226[2805] - Fixing APEX shutdown by explicitly shutting down throttling
- PR #4225[2806] - Allow `CMAKE_INSTALL_PREFIX` to be a relative path
- PR #4224[2807] - Deprecate verbs parcelport
- PR #4222[2808] - Update `register_{thread,work}` namespaces
- PR #4221[2809] - Changing `HPX_GCC_VERSION` check from `70000` to `70300`
- PR #4218[2810] - Google Season of Docs updates to documentation; grammar edits.
- PR #4217[2811] - Google Season of Docs updates to documentation; grammar edits.
- PR #4216[2812] - Fixing gcc warning on 32bit platforms (integer truncation)
- PR #4214[2813] - Apex callback refactoring
- PR #4213[2814] - Clean up allocator checks for dependent projects
- PR #4212[2815] - Google Season of Docs updates to documentation; grammar edits.
- PR #4211[2816] - Google Season of Docs updates to documentation; contributing to hpx
- PR #4210[2817] - Attempting to fix Intel compilation
- PR #4209[2818] - Fix CUDA 10 build
- PR #4205[2819] - Making sure that differences in `CMAKE_BUILD_TYPE` are not reported on multi-configuration cmake generators
- PR #4203[2820] - Deprecate Vc
- PR #4202[2821] - Fix CUDA configuration
- PR #4200[2822] - Making sure `hpx_wrap` is not passed on to linker on non-Linux systems
- PR #4198[2823] - Fix `execution_agent.cpp` compilation with GCC 5
- PR #4197[2824] - Remove deprecated options for 1.4.0 release

[2802] https://github.com/STEllAR-GROUP/hpx/pull/4234
[2803] https://github.com/STEllAR-GROUP/hpx/pull/4230
[2804] https://github.com/STEllAR-GROUP/hpx/pull/4229
[2805] https://github.com/STEllAR-GROUP/hpx/pull/4226
[2806] https://github.com/STEllAR-GROUP/hpx/pull/4225
[2807] https://github.com/STEllAR-GROUP/hpx/pull/4224
[2808] https://github.com/STEllAR-GROUP/hpx/pull/4222
[2809] https://github.com/STEllAR-GROUP/hpx/pull/4221
[2810] https://github.com/STEllAR-GROUP/hpx/pull/4218
[2811] https://github.com/STEllAR-GROUP/hpx/pull/4217
[2812] https://github.com/STEllAR-GROUP/hpx/pull/4216
[2813] https://github.com/STEllAR-GROUP/hpx/pull/4214
[2814] https://github.com/STEllAR-GROUP/hpx/pull/4213
[2815] https://github.com/STEllAR-GROUP/hpx/pull/4212
[2816] https://github.com/STEllAR-GROUP/hpx/pull/4211
[2817] https://github.com/STEllAR-GROUP/hpx/pull/4210
[2818] https://github.com/STEllAR-GROUP/hpx/pull/4209
[2819] https://github.com/STEllAR-GROUP/hpx/pull/4205
[2820] https://github.com/STEllAR-GROUP/hpx/pull/4203
[2821] https://github.com/STEllAR-GROUP/hpx/pull/4202
[2822] https://github.com/STEllAR-GROUP/hpx/pull/4200
[2823] https://github.com/STEllAR-GROUP/hpx/pull/4198
[2824] https://github.com/STEllAR-GROUP/hpx/pull/4197

- PR #4196[2825] - minor fixes for building on OSX Darwin
- PR #4195[2826] - Use full clone on CircleCI for pushing stable tag
- PR #4193[2827] - Add scheduling hints to hello_world_distributed
- PR #4192[2828] - Set up CUDA in HPXConfig.cmake
- PR #4191[2829] - Export allocators root variables
- PR #4190[2830] - Don't use `constexpr` in `thread_data` with GCC <= 6
- PR #4189[2831] - Only use `quick_exit` if available
- PR #4188[2832] - Google Season of Docs updates to documentation; writing single node hpx applications
- PR #4186[2833] - correct vc to cuda in cuda cmake
- PR #4184[2834] - Resetting some cached variables to make sure those are re-filled
- PR #4183[2835] - Fix `hpxcxx` configuration
- PR #4181[2836] - Rename base libraries var
- PR #4180[2837] - Move header left behind earlier to plugin module
- PR #4179[2838] - Moving `zip_iterator` and `transform_iterator` to iterator_support module
- PR #4178[2839] - Move checkpointing support to its own module
- PR #4177[2840] - Small const fix to `basic_execution` module
- PR #4176[2841] - Add back `HPX_LIBRARIES` and friends to `HPXConfig.cmake`
- PR #4175[2842] - Make Vc public and add it to `HPXConfig.cmake`
- PR #4173[2843] - Wait for runtime to be running before returning from hpx::start
- PR #4172[2844] - More protection against shutdown problems in error handling scenarios.
- PR #4171[2845] - Ignore lock in `condition_variable::wait`
- PR #4170[2846] - Adding APEX dependency to MPI parcelport
- PR #4168[2847] - Adding utility include

2825 https://github.com/STEllAR-GROUP/hpx/pull/4196
2826 https://github.com/STEllAR-GROUP/hpx/pull/4195
2827 https://github.com/STEllAR-GROUP/hpx/pull/4193
2828 https://github.com/STEllAR-GROUP/hpx/pull/4192
2829 https://github.com/STEllAR-GROUP/hpx/pull/4191
2830 https://github.com/STEllAR-GROUP/hpx/pull/4190
2831 https://github.com/STEllAR-GROUP/hpx/pull/4189
2832 https://github.com/STEllAR-GROUP/hpx/pull/4188
2833 https://github.com/STEllAR-GROUP/hpx/pull/4186
2834 https://github.com/STEllAR-GROUP/hpx/pull/4184
2835 https://github.com/STEllAR-GROUP/hpx/pull/4183
2836 https://github.com/STEllAR-GROUP/hpx/pull/4181
2837 https://github.com/STEllAR-GROUP/hpx/pull/4180
2838 https://github.com/STEllAR-GROUP/hpx/pull/4179
2839 https://github.com/STEllAR-GROUP/hpx/pull/4178
2840 https://github.com/STEllAR-GROUP/hpx/pull/4177
2841 https://github.com/STEllAR-GROUP/hpx/pull/4176
2842 https://github.com/STEllAR-GROUP/hpx/pull/4175
2843 https://github.com/STEllAR-GROUP/hpx/pull/4173
2844 https://github.com/STEllAR-GROUP/hpx/pull/4172
2845 https://github.com/STEllAR-GROUP/hpx/pull/4171
2846 https://github.com/STEllAR-GROUP/hpx/pull/4170
2847 https://github.com/STEllAR-GROUP/hpx/pull/4168

- PR #4167[2848] - Add a condition to setup the external libraries
- PR #4166[2849] - Add an `INTERNAL_FLAGS` option to link to `hpx_internal_flags`
- PR #4165[2850] - Forward `HPX_*` cmake cache variables to external projects
- PR #4164[2851] - Affinity and batch environment modules
- PR #4162[2852] - Handle `quick exit`
- PR #4160[2853] - Using `target_link_libraries` for cmake versions >= 3.12
- PR #4159[2854] - Make sure `HPX_WITH_NATIVE_TLS` is forwarded to dependent projects
- PR #4158[2855] - Adding allocator imported target as a dependency of allocator module
- PR #4157[2856] - Add `hpx_memory` as a dependency of parcelport plugins
- PR #4156[2857] - Stackless coroutines now can refer to themselves (through get_self() and friends)
- PR #4154[2858] - Added CMake policy CMP0060 for HPX applications.
- PR #4153[2859] - add header `iomanip` to tests and tool
- PR #4152[2860] - Casting MPI tag value
- PR #4149[2861] - Add back private `m_desc` member variable in program_options module
- PR #4147[2862] - Resource partitioner and threadmanager modules
- PR #4146[2863] - Google Season of Docs updates to documentation; creating hpx projects
- PR #4145[2864] - Adding basic support for stackless threads
- PR #4143[2865] - Exclude `test_client_1950` from all target
- PR #4142[2866] - Add a new `thread_pool_executor`
- PR #4140[2867] - Google Season of Docs updates to documentation; why hpx
- PR #4139[2868] - Remove runtime includes from coroutines module
- PR #4138[2869] - Forking `boost::intrusive_ptr` and adding it as `hpx::intrusive_ptr`
- PR #4137[2870] - Fixing TSS destruction

[2848] https://github.com/STEllAR-GROUP/hpx/pull/4167
[2849] https://github.com/STEllAR-GROUP/hpx/pull/4166
[2850] https://github.com/STEllAR-GROUP/hpx/pull/4165
[2851] https://github.com/STEllAR-GROUP/hpx/pull/4164
[2852] https://github.com/STEllAR-GROUP/hpx/pull/4162
[2853] https://github.com/STEllAR-GROUP/hpx/pull/4160
[2854] https://github.com/STEllAR-GROUP/hpx/pull/4159
[2855] https://github.com/STEllAR-GROUP/hpx/pull/4158
[2856] https://github.com/STEllAR-GROUP/hpx/pull/4157
[2857] https://github.com/STEllAR-GROUP/hpx/pull/4156
[2858] https://github.com/STEllAR-GROUP/hpx/pull/4154
[2859] https://github.com/STEllAR-GROUP/hpx/pull/4153
[2860] https://github.com/STEllAR-GROUP/hpx/pull/4152
[2861] https://github.com/STEllAR-GROUP/hpx/pull/4149
[2862] https://github.com/STEllAR-GROUP/hpx/pull/4147
[2863] https://github.com/STEllAR-GROUP/hpx/pull/4146
[2864] https://github.com/STEllAR-GROUP/hpx/pull/4145
[2865] https://github.com/STEllAR-GROUP/hpx/pull/4143
[2866] https://github.com/STEllAR-GROUP/hpx/pull/4142
[2867] https://github.com/STEllAR-GROUP/hpx/pull/4140
[2868] https://github.com/STEllAR-GROUP/hpx/pull/4139
[2869] https://github.com/STEllAR-GROUP/hpx/pull/4138
[2870] https://github.com/STEllAR-GROUP/hpx/pull/4137

- PR #4136[2871] - HPX.Compute modules
- PR #4133[2872] - Fix `block_executor`
- PR #4131[2873] - Applying fixes based on reports from PVS Studio
- PR #4130[2874] - Adding missing header to build system
- PR #4129[2875] - Fixing compilation if `HPX_WITH_DATAPAR_VC` is enabled
- PR #4128[2876] - Renaming `moveonly_any` to `unique_any`
- PR #4126[2877] - Attempt to fix `basic_any` constructor for gcc 7
- PR #4125[2878] - Changing `extra_archive_data` implementation
- PR #4124[2879] - Don't link to Boost.System unless required
- PR #4122[2880] - Add kernel launch helper utility (+saxpy demo) and merge in octotiger changes
- PR #4121[2881] - Fixing migration test if networking is disabled.
- PR #4120[2882] - Google Season of Docs updates to documentation; hpx build system v1
- PR #4119[2883] - Making sure `chunk_size` and `max_chunk` are actually applied to parallel algorithms if specified
- PR #4117[2884] - Make CircleCI formatting check store diff
- PR #4116[2885] - Fix automatically setting C++ standard
- PR #4114[2886] - Module serialization
- PR #4113[2887] - Module datastructures
- PR #4111[2888] - Fixing performance regression introduced earlier
- PR #4110[2889] - Adding missing SPDX tags
- PR #4109[2890] - Overload for start without entry point/argv.
- PR #4108[2891] - Making sure C++ standard is properly detected and propagated
- PR #4106[2892] - use `std::round` for guaranteed rounding without errors
- PR #4104[2893] - Extend `scheduler_mode` with new `work_stealing` and task assignment modes

[2871] https://github.com/STEllAR-GROUP/hpx/pull/4136
[2872] https://github.com/STEllAR-GROUP/hpx/pull/4133
[2873] https://github.com/STEllAR-GROUP/hpx/pull/4131
[2874] https://github.com/STEllAR-GROUP/hpx/pull/4130
[2875] https://github.com/STEllAR-GROUP/hpx/pull/4129
[2876] https://github.com/STEllAR-GROUP/hpx/pull/4128
[2877] https://github.com/STEllAR-GROUP/hpx/pull/4126
[2878] https://github.com/STEllAR-GROUP/hpx/pull/4125
[2879] https://github.com/STEllAR-GROUP/hpx/pull/4124
[2880] https://github.com/STEllAR-GROUP/hpx/pull/4122
[2881] https://github.com/STEllAR-GROUP/hpx/pull/4121
[2882] https://github.com/STEllAR-GROUP/hpx/pull/4120
[2883] https://github.com/STEllAR-GROUP/hpx/pull/4119
[2884] https://github.com/STEllAR-GROUP/hpx/pull/4117
[2885] https://github.com/STEllAR-GROUP/hpx/pull/4116
[2886] https://github.com/STEllAR-GROUP/hpx/pull/4114
[2887] https://github.com/STEllAR-GROUP/hpx/pull/4113
[2888] https://github.com/STEllAR-GROUP/hpx/pull/4111
[2889] https://github.com/STEllAR-GROUP/hpx/pull/4110
[2890] https://github.com/STEllAR-GROUP/hpx/pull/4109
[2891] https://github.com/STEllAR-GROUP/hpx/pull/4108
[2892] https://github.com/STEllAR-GROUP/hpx/pull/4106
[2893] https://github.com/STEllAR-GROUP/hpx/pull/4104

- PR #4103[2894] - Add this to lambda capture list
- PR #4102[2895] - Add spdx license and check
- PR #4099[2896] - Module coroutines
- PR #4098[2897] - Fix append module path in module CMakeLists template
- PR #4097[2898] - Function tests
- PR #4096[2899] - Removing return of `thread_result_type` from functions not needing them
- PR #4095[2900] - Stop-gap measure until cmake overhaul is in place
- PR #4094[2901] - Deprecate `HPX_WITH_MORE_THAN_64_THREADS`
- PR #4093[2902] - Fix initialization of `global_num_tasks` in `parallel_executor`
- PR #4092[2903] - Add support for mi-malloc
- PR #4090[2904] - Execution context
- PR #4089[2905] - Make counters in coroutines optional
- PR #4087[2906] - Making `hpx::util::any` compatible with C++17
- PR #4084[2907] - Making sure destination array for `std::transform` is properly resized
- PR #4083[2908] - Adapting `thread_queue_mc` to behave even if no 128bit atomics are available
- PR #4082[2909] - Fix compilation on GCC 5
- PR #4081[2910] - Adding option allowing to force using Boost.FileSystem
- PR #4080[2911] - Updating module dependencies
- PR #4079[2912] - Add missing tests for iterator_support module
- PR #4078[2913] - Disable parcel-layer if networking is disabled
- PR #4077[2914] - Add missing include that causes build fails
- PR #4076[2915] - Enable compatibility headers for functional module
- PR #4075[2916] - Coroutines module

[2894] https://github.com/STEllAR-GROUP/hpx/pull/4103
[2895] https://github.com/STEllAR-GROUP/hpx/pull/4102
[2896] https://github.com/STEllAR-GROUP/hpx/pull/4099
[2897] https://github.com/STEllAR-GROUP/hpx/pull/4098
[2898] https://github.com/STEllAR-GROUP/hpx/pull/4097
[2899] https://github.com/STEllAR-GROUP/hpx/pull/4096
[2900] https://github.com/STEllAR-GROUP/hpx/pull/4095
[2901] https://github.com/STEllAR-GROUP/hpx/pull/4094
[2902] https://github.com/STEllAR-GROUP/hpx/pull/4093
[2903] https://github.com/STEllAR-GROUP/hpx/pull/4092
[2904] https://github.com/STEllAR-GROUP/hpx/pull/4090
[2905] https://github.com/STEllAR-GROUP/hpx/pull/4089
[2906] https://github.com/STEllAR-GROUP/hpx/pull/4087
[2907] https://github.com/STEllAR-GROUP/hpx/pull/4084
[2908] https://github.com/STEllAR-GROUP/hpx/pull/4083
[2909] https://github.com/STEllAR-GROUP/hpx/pull/4082
[2910] https://github.com/STEllAR-GROUP/hpx/pull/4081
[2911] https://github.com/STEllAR-GROUP/hpx/pull/4080
[2912] https://github.com/STEllAR-GROUP/hpx/pull/4079
[2913] https://github.com/STEllAR-GROUP/hpx/pull/4078
[2914] https://github.com/STEllAR-GROUP/hpx/pull/4077
[2915] https://github.com/STEllAR-GROUP/hpx/pull/4076
[2916] https://github.com/STEllAR-GROUP/hpx/pull/4075

- PR #4073[2917] - Use `configure_file` for generated files in modules
- PR #4071[2918] - Fixing MPI detection for PMIx
- PR #4070[2919] - Fix macOS builds
- PR #4069[2920] - Moving more facilities to the collectives module
- PR #4068[2921] - Adding main HPX `#include` directory to modules
- PR #4066[2922] - Switching the use of `message(STATUS "...")` to hpx_info
- PR #4065[2923] - Move Boost.Filesystem handling to filesystem module
- PR #4064[2924] - Fix program_options test with older boost versions
- PR #4062[2925] - The `cpu_features` tool fails to compile on anything but x86 architectures
- PR #4061[2926] - Add `clang-format` checking step for modules
- PR #4060[2927] - Making sure `HPX_IDLE_BACKOFF_TIME_MAX` is always defined (even if its unused)
- PR #4059[2928] - Renaming module `hpx_parallel_executors` into `hpx_execution`
- PR #4058[2929] - Do not build networking tests when networking disabled
- PR #4057[2930] - Printing configuration summary for modules as well
- PR #4055[2931] - Google Season of Docs updates to documentation; hpx build systems
- PR #4054[2932] - Add troubleshooting section to manual
- PR #4051[2933] - Add more variations to `future_overhead` test
- PR #4050[2934] - Creating plugin module
- PR #4049[2935] - Move missing modules tests
- PR #4047[2936] - Add boost/filesystem headers to inspect deprecated headers
- PR #4045[2937] - Module functional
- PR #4043[2938] - Fix preconditions and error messages for suspension functions
- PR #4041[2939] - Pass HPX_STANDARD on to dependent projects via HPXConfig.cmake

2917 https://github.com/STEllAR-GROUP/hpx/pull/4073
2918 https://github.com/STEllAR-GROUP/hpx/pull/4071
2919 https://github.com/STEllAR-GROUP/hpx/pull/4070
2920 https://github.com/STEllAR-GROUP/hpx/pull/4069
2921 https://github.com/STEllAR-GROUP/hpx/pull/4068
2922 https://github.com/STEllAR-GROUP/hpx/pull/4066
2923 https://github.com/STEllAR-GROUP/hpx/pull/4065
2924 https://github.com/STEllAR-GROUP/hpx/pull/4064
2925 https://github.com/STEllAR-GROUP/hpx/pull/4062
2926 https://github.com/STEllAR-GROUP/hpx/pull/4061
2927 https://github.com/STEllAR-GROUP/hpx/pull/4060
2928 https://github.com/STEllAR-GROUP/hpx/pull/4059
2929 https://github.com/STEllAR-GROUP/hpx/pull/4058
2930 https://github.com/STEllAR-GROUP/hpx/pull/4057
2931 https://github.com/STEllAR-GROUP/hpx/pull/4055
2932 https://github.com/STEllAR-GROUP/hpx/pull/4054
2933 https://github.com/STEllAR-GROUP/hpx/pull/4051
2934 https://github.com/STEllAR-GROUP/hpx/pull/4050
2935 https://github.com/STEllAR-GROUP/hpx/pull/4049
2936 https://github.com/STEllAR-GROUP/hpx/pull/4047
2937 https://github.com/STEllAR-GROUP/hpx/pull/4045
2938 https://github.com/STEllAR-GROUP/hpx/pull/4043
2939 https://github.com/STEllAR-GROUP/hpx/pull/4041

- PR #4040[2940] - Program options module
- PR #4039[2941] - Moving non-serializable `any` (`any_nonser`) to datastructures module
- PR #4038[2942] - Adding MPark's variant (V1.4.0) to HPX
- PR #4037[2943] - Adding resiliency module
- PR #4036[2944] - Add C++17 filesystem compatibility header
- PR #4035[2945] - Fixing support for mpirun
- PR #4028[2946] - CMake to target based directives
- PR #4027[2947] - Remove GitLab CI configuration
- PR #4026[2948] - Threading refactoring
- PR #4025[2949] - Refactoring thread queue configuration options
- PR #4024[2950] - Fix padding calculation in `cache_aligned_data.hpp`
- PR #4023[2951] - Fixing Codacy issues
- PR #4022[2952] - Make sure process mask option is passed to `affinity_data`
- PR #4021[2953] - Warn about compiling in C++11 mode
- PR #4020[2954] - Module concurrency
- PR #4019[2955] - Module topology
- PR #4018[2956] - Update deprecated header in `thread_queue_mc.hpp`
- PR #4015[2957] - Avoid overwriting artifacts
- PR #4014[2958] - Future overheads
- PR #4013[2959] - Update URL to test output conversion script
- PR #4012[2960] - Fix CUDA compilation
- PR #4011[2961] - Fixing cyclic dependencies between modules
- PR #4010[2962] - Ignore stable tag on CircleCI

[2940] https://github.com/STEllAR-GROUP/hpx/pull/4040
[2941] https://github.com/STEllAR-GROUP/hpx/pull/4039
[2942] https://github.com/STEllAR-GROUP/hpx/pull/4038
[2943] https://github.com/STEllAR-GROUP/hpx/pull/4037
[2944] https://github.com/STEllAR-GROUP/hpx/pull/4036
[2945] https://github.com/STEllAR-GROUP/hpx/pull/4035
[2946] https://github.com/STEllAR-GROUP/hpx/pull/4028
[2947] https://github.com/STEllAR-GROUP/hpx/pull/4027
[2948] https://github.com/STEllAR-GROUP/hpx/pull/4026
[2949] https://github.com/STEllAR-GROUP/hpx/pull/4025
[2950] https://github.com/STEllAR-GROUP/hpx/pull/4024
[2951] https://github.com/STEllAR-GROUP/hpx/pull/4023
[2952] https://github.com/STEllAR-GROUP/hpx/pull/4022
[2953] https://github.com/STEllAR-GROUP/hpx/pull/4021
[2954] https://github.com/STEllAR-GROUP/hpx/pull/4020
[2955] https://github.com/STEllAR-GROUP/hpx/pull/4019
[2956] https://github.com/STEllAR-GROUP/hpx/pull/4018
[2957] https://github.com/STEllAR-GROUP/hpx/pull/4015
[2958] https://github.com/STEllAR-GROUP/hpx/pull/4014
[2959] https://github.com/STEllAR-GROUP/hpx/pull/4013
[2960] https://github.com/STEllAR-GROUP/hpx/pull/4012
[2961] https://github.com/STEllAR-GROUP/hpx/pull/4011
[2962] https://github.com/STEllAR-GROUP/hpx/pull/4010

- PR #4009[2963] - Check circular dependencies in a circle ci step
- PR #4008[2964] - Extend cache aligned data to handle tuple-like data
- PR #4007[2965] - Fixing migration for components that have actions returning a client
- PR #4006[2966] - Move is_value_proxy.hpp to algorithms module
- PR #4004[2967] - Shorten CTest timeout on CircleCI
- PR #4003[2968] - Refactoring to remove (internal) dependencies
- PR #4001[2969] - Exclude tests from all target
- PR #4000[2970] - Module errors
- PR #3999[2971] - Enable support for compatibility headers for logging module
- PR #3998[2972] - Add process thread binding option
- PR #3997[2973] - Export handle_assert function
- PR #3996[2974] - Attempt to solve issue where `-latomic` does not support 128bit atomics
- PR #3993[2975] - Make sure `__LINE__` is an unsigned
- PR #3991[2976] - Fix dependencies and flags for header tests
- PR #3990[2977] - Documentation tags fixes
- PR #3988[2978] - Adding missing solution folder for format module test
- PR #3987[2979] - Move runtime-dependent functions out of command line handling
- PR #3986[2980] - Fix CMake configuration with PAPI on
- PR #3985[2981] - Module timing
- PR #3984[2982] - Fix default behaviour of paths in `add_hpx_component`
- PR #3982[2983] - Parallel executors module
- PR #3981[2984] - Segmented algorithms module
- PR #3980[2985] - Module logging

2963 https://github.com/STEllAR-GROUP/hpx/pull/4009
2964 https://github.com/STEllAR-GROUP/hpx/pull/4008
2965 https://github.com/STEllAR-GROUP/hpx/pull/4007
2966 https://github.com/STEllAR-GROUP/hpx/pull/4006
2967 https://github.com/STEllAR-GROUP/hpx/pull/4004
2968 https://github.com/STEllAR-GROUP/hpx/pull/4003
2969 https://github.com/STEllAR-GROUP/hpx/pull/4001
2970 https://github.com/STEllAR-GROUP/hpx/pull/4000
2971 https://github.com/STEllAR-GROUP/hpx/pull/3999
2972 https://github.com/STEllAR-GROUP/hpx/pull/3998
2973 https://github.com/STEllAR-GROUP/hpx/pull/3997
2974 https://github.com/STEllAR-GROUP/hpx/pull/3996
2975 https://github.com/STEllAR-GROUP/hpx/pull/3993
2976 https://github.com/STEllAR-GROUP/hpx/pull/3991
2977 https://github.com/STEllAR-GROUP/hpx/pull/3990
2978 https://github.com/STEllAR-GROUP/hpx/pull/3988
2979 https://github.com/STEllAR-GROUP/hpx/pull/3987
2980 https://github.com/STEllAR-GROUP/hpx/pull/3986
2981 https://github.com/STEllAR-GROUP/hpx/pull/3985
2982 https://github.com/STEllAR-GROUP/hpx/pull/3984
2983 https://github.com/STEllAR-GROUP/hpx/pull/3982
2984 https://github.com/STEllAR-GROUP/hpx/pull/3981
2985 https://github.com/STEllAR-GROUP/hpx/pull/3980

- PR #3979[2986] - Module util
- PR #3978[2987] - Fix `clang-tidy` step on CircleCI
- PR #3977[2988] - Fixing solution folders for moved components
- PR #3976[2989] - Module format
- PR #3975[2990] - Enable deprecation warnings on CircleCI
- PR #3974[2991] - Fix typos in documentation
- PR #3973[2992] - Fix compilation with GCC 9
- PR #3972[2993] - Add condition to clone apex + use of new cmake var APEX_ROOT
- PR #3971[2994] - Add testing module
- PR #3968[2995] - Remove unneeded file in hardware module
- PR #3967[2996] - Remove leftover PIC settings from main CMakeLists.txt
- PR #3966[2997] - Add missing export option in `add_hpx_module`
- PR #3965[2998] - Change `current_function_helper` back to non-constexpr
- PR #3964[2999] - Fixing merge problems
- PR #3962[3000] - Add a trait for `std::array` for unwrapping
- PR #3961[3001] - Making `hpx::util::tuple<Ts...>` and `std::tuple<Ts...>` convertible
- PR #3960[3002] - fix compilation with CUDA 10 and GCC 6
- PR #3959[3003] - Fix C++11 incompatibility
- PR #3957[3004] - Algorithms module
- PR #3956[3005] - [`HPX_AddModule`] Fix lower name var to upper
- PR #3955[3006] - Fix CMake configuration with examples off and tests on
- PR #3954[3007] - Move components to separate subdirectory in root of repository
- PR #3952[3008] - Update `papi.cpp`

2986 https://github.com/STEllAR-GROUP/hpx/pull/3979
2987 https://github.com/STEllAR-GROUP/hpx/pull/3978
2988 https://github.com/STEllAR-GROUP/hpx/pull/3977
2989 https://github.com/STEllAR-GROUP/hpx/pull/3976
2990 https://github.com/STEllAR-GROUP/hpx/pull/3975
2991 https://github.com/STEllAR-GROUP/hpx/pull/3974
2992 https://github.com/STEllAR-GROUP/hpx/pull/3973
2993 https://github.com/STEllAR-GROUP/hpx/pull/3972
2994 https://github.com/STEllAR-GROUP/hpx/pull/3971
2995 https://github.com/STEllAR-GROUP/hpx/pull/3968
2996 https://github.com/STEllAR-GROUP/hpx/pull/3967
2997 https://github.com/STEllAR-GROUP/hpx/pull/3966
2998 https://github.com/STEllAR-GROUP/hpx/pull/3965
2999 https://github.com/STEllAR-GROUP/hpx/pull/3964
3000 https://github.com/STEllAR-GROUP/hpx/pull/3962
3001 https://github.com/STEllAR-GROUP/hpx/pull/3961
3002 https://github.com/STEllAR-GROUP/hpx/pull/3960
3003 https://github.com/STEllAR-GROUP/hpx/pull/3959
3004 https://github.com/STEllAR-GROUP/hpx/pull/3957
3005 https://github.com/STEllAR-GROUP/hpx/pull/3956
3006 https://github.com/STEllAR-GROUP/hpx/pull/3955
3007 https://github.com/STEllAR-GROUP/hpx/pull/3954
3008 https://github.com/STEllAR-GROUP/hpx/pull/3952

- PR #3951[3009] - Exclude modules header tests from all target
- PR #3950[3010] - Adding `all_reduce` facility to collectives module
- PR #3949[3011] - This adds a configuration file that will cause for stale issues to be automatically closed
- PR #3948[3012] - Fixing ALPS environment
- PR #3947[3013] - Add major compiler version check for building hpx as a binary package
- PR #3946[3014] - [Modules] Move the location of the generated headers
- PR #3945[3015] - Simplify tests and examples cmake
- PR #3943[3016] - Remove example module
- PR #3942[3017] - Add `NOEXPORT` option to `add_hpx_{component,library}`
- PR #3938[3018] - Use https for CDash submissions
- PR #3937[3019] - Add `HPX_WITH_BUILD_BINARY_PACKAGE` to the compiler check (refs #3935)
- PR #3936[3020] - Fixing installation of binaries on windows
- PR #3934[3021] - Add set function for `sliding_semaphore` `max_difference`
- PR #3933[3022] - Remove `cudadevrt` from compile/link flags as it breaks downstream projects
- PR #3932[3023] - Fixing 3929
- PR #3931[3024] - Adding `all_to_all`
- PR #3930[3025] - Add test demonstrating the use of broadcast with component actions
- PR #3928[3026] - fixed number of tasks and number of threads for heterogeneous slurm environments
- PR #3927[3027] - Moving Cache module's tests into separate solution folder
- PR #3926[3028] - Move unit tests to cache module
- PR #3925[3029] - Move version check to config module
- PR #3924[3030] - Add schedule hint executor parameters
- PR #3923[3031] - Allow aligning objects bigger than the cache line size

3009 https://github.com/STEllAR-GROUP/hpx/pull/3951
3010 https://github.com/STEllAR-GROUP/hpx/pull/3950
3011 https://github.com/STEllAR-GROUP/hpx/pull/3949
3012 https://github.com/STEllAR-GROUP/hpx/pull/3948
3013 https://github.com/STEllAR-GROUP/hpx/pull/3947
3014 https://github.com/STEllAR-GROUP/hpx/pull/3946
3015 https://github.com/STEllAR-GROUP/hpx/pull/3945
3016 https://github.com/STEllAR-GROUP/hpx/pull/3943
3017 https://github.com/STEllAR-GROUP/hpx/pull/3942
3018 https://github.com/STEllAR-GROUP/hpx/pull/3938
3019 https://github.com/STEllAR-GROUP/hpx/pull/3937
3020 https://github.com/STEllAR-GROUP/hpx/pull/3936
3021 https://github.com/STEllAR-GROUP/hpx/pull/3934
3022 https://github.com/STEllAR-GROUP/hpx/pull/3933
3023 https://github.com/STEllAR-GROUP/hpx/pull/3932
3024 https://github.com/STEllAR-GROUP/hpx/pull/3931
3025 https://github.com/STEllAR-GROUP/hpx/pull/3930
3026 https://github.com/STEllAR-GROUP/hpx/pull/3928
3027 https://github.com/STEllAR-GROUP/hpx/pull/3927
3028 https://github.com/STEllAR-GROUP/hpx/pull/3926
3029 https://github.com/STEllAR-GROUP/hpx/pull/3925
3030 https://github.com/STEllAR-GROUP/hpx/pull/3924
3031 https://github.com/STEllAR-GROUP/hpx/pull/3923

- PR #3922[3032] - Add Windows builds with Travis CI
- PR #3921[3033] - Add ccls cache directory to gitignore
- PR #3920[3034] - Fix `git_external` fetching of tags
- PR #3905[3035] - Correct rostambod url. Fix typo in doc
- PR #3904[3036] - Fix bug in context_base.hpp
- PR #3903[3037] - Adding new performance counters
- PR #3902[3038] - Add `add_hpx_module` function
- PR #3901[3039] - Factoring out container remapping into a separate trait
- PR #3900[3040] - Making sure errors during command line processing are properly reported and will not cause assertions
- PR #3899[3041] - Remove old compatibility bases from `make_action`
- PR #3898[3042] - Make parameter size be of type `size_t`
- PR #3897[3043] - Making sure all tests are disabled if `HPX_WITH_TESTS=OFF`
- PR #3895[3044] - Add documentation for annotated_function
- PR #3894[3045] - Working around VS2019 problem with `make_action`
- PR #3892[3046] - Avoid MSVC compatibility warning in internal allocator
- PR #3891[3047] - Removal of the default intel config include
- PR #3888[3048] - Fix `async_customization` dataflow example and Clarify what's being tested
- PR #3887[3049] - Add Doxygen documentation
- PR #3882[3050] - Minor docs fixes
- PR #3880[3051] - Updating APEX version tag
- PR #3878[3052] - Making sure symbols are properly exported from modules (needed for Windows/MacOS)
- PR #3877[3053] - Documentation
- PR #3876[3054] - Module hardware

[3032] https://github.com/STEllAR-GROUP/hpx/pull/3922
[3033] https://github.com/STEllAR-GROUP/hpx/pull/3921
[3034] https://github.com/STEllAR-GROUP/hpx/pull/3920
[3035] https://github.com/STEllAR-GROUP/hpx/pull/3905
[3036] https://github.com/STEllAR-GROUP/hpx/pull/3904
[3037] https://github.com/STEllAR-GROUP/hpx/pull/3903
[3038] https://github.com/STEllAR-GROUP/hpx/pull/3902
[3039] https://github.com/STEllAR-GROUP/hpx/pull/3901
[3040] https://github.com/STEllAR-GROUP/hpx/pull/3900
[3041] https://github.com/STEllAR-GROUP/hpx/pull/3899
[3042] https://github.com/STEllAR-GROUP/hpx/pull/3898
[3043] https://github.com/STEllAR-GROUP/hpx/pull/3897
[3044] https://github.com/STEllAR-GROUP/hpx/pull/3895
[3045] https://github.com/STEllAR-GROUP/hpx/pull/3894
[3046] https://github.com/STEllAR-GROUP/hpx/pull/3892
[3047] https://github.com/STEllAR-GROUP/hpx/pull/3891
[3048] https://github.com/STEllAR-GROUP/hpx/pull/3888
[3049] https://github.com/STEllAR-GROUP/hpx/pull/3887
[3050] https://github.com/STEllAR-GROUP/hpx/pull/3882
[3051] https://github.com/STEllAR-GROUP/hpx/pull/3880
[3052] https://github.com/STEllAR-GROUP/hpx/pull/3878
[3053] https://github.com/STEllAR-GROUP/hpx/pull/3877
[3054] https://github.com/STEllAR-GROUP/hpx/pull/3876

- PR #3875[3055] - Converted typedefs in actions submodule to using directives
- PR #3874[3056] - Allow one to suppress target keywords in `hpx_setup_target` for backwards compatibility
- PR #3873[3057] - Add scripts to create releases and generate lists of PRs and issues
- PR #3872[3058] - Fix latest HTML docs location
- PR #3870[3059] - Module cache
- PR #3869[3060] - Post 1.3.0 version bumps
- PR #3868[3061] - Replace the macro `HPX_ASSERT` by `HPX_TEST` in tests
- PR #3845[3062] - Assertion module
- PR #3839[3063] - Make tuple serialization non-intrusive
- PR #3832[3064] - Config module
- PR #3799[3065] - Remove compat namespace and its contents
- PR #3701[3066] - MoodyCamel lockfree
- PR #3496[3067] - Disabling MPI's (deprecated) C++ interface
- PR #3192[3068] - Move type info into `hpx::debug` namespace and add print helper functions
- PR #3159[3069] - Support Checkpointing Components

HPX V1.3.0 (May 23, 2019)

General changes

- Performance improvements: the schedulers have significantly reduced overheads from removing false sharing and the parallel executor has been updated to create fewer futures.
- HPX now defaults to not turning on networking when running on one locality. This means that you can run multiple instances on the same system without adding command line options.
- Multiple issues reported by Clang sanitizers have been fixed.
- We have added (back) single-page HTML documentation and PDF documentation.
- We have started modularizing the HPX library. This is useful both for developers and users. In the long term users will be able to consume only parts of the HPX libraries if they do not require all the functionality that HPX currently provides.
- We have added an implementation of `function_ref`.

[3055] https://github.com/STEllAR-GROUP/hpx/pull/3875
[3056] https://github.com/STEllAR-GROUP/hpx/pull/3874
[3057] https://github.com/STEllAR-GROUP/hpx/pull/3873
[3058] https://github.com/STEllAR-GROUP/hpx/pull/3872
[3059] https://github.com/STEllAR-GROUP/hpx/pull/3870
[3060] https://github.com/STEllAR-GROUP/hpx/pull/3869
[3061] https://github.com/STEllAR-GROUP/hpx/pull/3868
[3062] https://github.com/STEllAR-GROUP/hpx/pull/3845
[3063] https://github.com/STEllAR-GROUP/hpx/pull/3839
[3064] https://github.com/STEllAR-GROUP/hpx/pull/3832
[3065] https://github.com/STEllAR-GROUP/hpx/pull/3799
[3066] https://github.com/STEllAR-GROUP/hpx/pull/3701
[3067] https://github.com/STEllAR-GROUP/hpx/pull/3496
[3068] https://github.com/STEllAR-GROUP/hpx/pull/3192
[3069] https://github.com/STEllAR-GROUP/hpx/pull/3159

- The `barrier` and `latch` classes have gained a few additional member functions.

Breaking changes

- Executable and library targets are now created without the `_exe` and `_lib` suffix respectively. For example, the target `1d_stencil_1_exe` is now simply called `1d_stencil_1`.
- We have removed the following deprecated functionality: `queue`, `scoped_unlock`, and support for input iterators in algorithms.
- We have turned off the compatibility layer for `unwrapped` by default. The functionality will be removed in the next release. The option can still be turned on using the CMake[3070] option `HPX_WITH_UNWRAPPED_SUPPORT`. Likewise, `inclusive_scan` compatibility overloads have been turned off by default. They can still be turned on with `HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY`.
- The minimum compiler and dependency versions have been updated. We now support GCC from version 5 onwards, Clang from version 4 onwards, and Boost from version 1.61.0 onwards.
- The headers for preprocessor macros have moved as a result of the functionality being moved to a separate module. The old headers are deprecated and will be removed in a future version of HPX. You can turn off the warnings by setting `HPX_PREPROCESSOR_WITH_DEPRECATION_WARNINGS=OFF` or turn off the compatibility headers completely with `HPX_PREPROCESSOR_WITH_COMPATIBILITY_HEADERS=OFF`.

Closed issues

- Issue #3863[3071] - shouldn't "-faligned-new" be a usage requirement?
- Issue #3841[3072] - Build error with msvc 19 caused by SFINAE and C++17
- Issue #3836[3073] - master branch does not build with idle rate counters enabled
- Issue #3819[3074] - Add debug suffix to modules built in debug mode
- Issue #3817[3075] - `HPX_INCLUDE_DIRS` contains non-existent directory
- Issue #3810[3076] - Source groups are not created for files in modules
- Issue #3805[3077] - HPX won't compile with `-DHPX_WITH_APEX=TRUE`
- Issue #3792[3078] - Barrier Hangs When Locality Zero not included
- Issue #3778[3079] - Replace `throw()` with `noexcept`
- Issue #3763[3080] - configurable sort limit per task
- Issue #3758[3081] - dataflow doesn't convert `future<future<T>>` to `future<T>`

[3070] https://www.cmake.org
[3071] https://github.com/STEllAR-GROUP/hpx/issues/3863
[3072] https://github.com/STEllAR-GROUP/hpx/issues/3841
[3073] https://github.com/STEllAR-GROUP/hpx/issues/3836
[3074] https://github.com/STEllAR-GROUP/hpx/issues/3819
[3075] https://github.com/STEllAR-GROUP/hpx/issues/3817
[3076] https://github.com/STEllAR-GROUP/hpx/issues/3810
[3077] https://github.com/STEllAR-GROUP/hpx/issues/3805
[3078] https://github.com/STEllAR-GROUP/hpx/issues/3792
[3079] https://github.com/STEllAR-GROUP/hpx/issues/3778
[3080] https://github.com/STEllAR-GROUP/hpx/issues/3763
[3081] https://github.com/STEllAR-GROUP/hpx/issues/3758

- Issue #3757[3082] - When compiling undefined reference to `hpx::hpx_check_version_1_2` HPX V1.2.1, Ubuntu 18.04.01 Server Edition
- Issue #3753[3083] - `--hpx:list-counters=full` crashes
- Issue #3746[3084] - Detection of MPI with pmix
- Issue #3744[3085] - Separate spinlock from same cacheline as internal data for all LCOs
- Issue #3743[3086] - hpxcxx's shebang doesn't specify the python version
- Issue #3738[3087] - Unable to debug parcelport on a single node
- Issue #3735[3088] - Latest master: Can't compile in MSVC
- Issue #3731[3089] - `util::bound` seems broken on Clang with older libstdc++
- Issue #3724[3090] - Allow to pre-set command line options through environment
- Issue #3723[3091] - examples/resource_partitioner build issue on master branch / ubuntu 18
- Issue #3721[3092] - faced a building error
- Issue #3720[3093] - Hello World example fails to link
- Issue #3719[3094] - pkg-config produces invalid output: `-l-pthread`
- Issue #3718[3095] - Please make the python executable configurable through cmake
- Issue #3717[3096] - interested to contribute to the organisation
- Issue #3699[3097] - Remove 'HPX runtime' executable
- Issue #3698[3098] - Ignore all locks while handling asserts
- Issue #3689[3099] - Incorrect and inconsistent website structure http://stellar.cct.lsu.edu/downloads/.
- Issue #3681[3100] - Broken links on http://stellar.cct.lsu.edu/2015/05/hpx-archives-now-on-gmane/
- Issue #3676[3101] - HPX master built from source, cmake fails to link main.cpp example in docs
- Issue #3673[3102] - HPX build fails with `std::atomic` missing error
- Issue #3670[3103] - Generate PDF again from documentation (with Sphinx)
- Issue #3643[3104] - Warnings when compiling HPX 1.2.1 with gcc 9

[3082] https://github.com/STEllAR-GROUP/hpx/issues/3757
[3083] https://github.com/STEllAR-GROUP/hpx/issues/3753
[3084] https://github.com/STEllAR-GROUP/hpx/issues/3746
[3085] https://github.com/STEllAR-GROUP/hpx/issues/3744
[3086] https://github.com/STEllAR-GROUP/hpx/issues/3743
[3087] https://github.com/STEllAR-GROUP/hpx/issues/3738
[3088] https://github.com/STEllAR-GROUP/hpx/issues/3735
[3089] https://github.com/STEllAR-GROUP/hpx/issues/3731
[3090] https://github.com/STEllAR-GROUP/hpx/issues/3724
[3091] https://github.com/STEllAR-GROUP/hpx/issues/3723
[3092] https://github.com/STEllAR-GROUP/hpx/issues/3721
[3093] https://github.com/STEllAR-GROUP/hpx/issues/3720
[3094] https://github.com/STEllAR-GROUP/hpx/issues/3719
[3095] https://github.com/STEllAR-GROUP/hpx/issues/3718
[3096] https://github.com/STEllAR-GROUP/hpx/issues/3717
[3097] https://github.com/STEllAR-GROUP/hpx/issues/3699
[3098] https://github.com/STEllAR-GROUP/hpx/issues/3698
[3099] https://github.com/STEllAR-GROUP/hpx/issues/3689
[3100] https://github.com/STEllAR-GROUP/hpx/issues/3681
[3101] https://github.com/STEllAR-GROUP/hpx/issues/3676
[3102] https://github.com/STEllAR-GROUP/hpx/issues/3673
[3103] https://github.com/STEllAR-GROUP/hpx/issues/3670
[3104] https://github.com/STEllAR-GROUP/hpx/issues/3643

- Issue #3641[3105] - Trouble with using ranges-v3 and `hpx::parallel::reduce`
- Issue #3639[3106] - `util::unwrapping` does not work well with member functions
- Issue #3634[3107] - The build fails if `shared_future<>::then` is called with a thread executor
- Issue #3622[3108] - VTune Amplifier 2019 not working with `use_itt_notify=1`
- Issue #3616[3109] - HPX Fails to Build with CUDA 10
- Issue #3612[3110] - False sharing of scheduling counters
- Issue #3609[3111] - executor_parameters timeout with gcc <= 7 and Debug mode
- Issue #3601[3112] - Misleading error message on power pc for rdtsc and rdtscp
- Issue #3598[3113] - Build of some examples fails when using Vc
- Issue #3594[3114] - Error: The number of OS threads requested (20) does not match the number of threads to bind (12): HPX(bad_parameter)
- Issue #3592[3115] - Undefined Reference Error
- Issue #3589[3116] - include could not find load file: HPX_Utils.cmake
- Issue #3587[3117] - HPX won't compile on POWER8 with Clang 7
- Issue #3583[3118] - Fedora and openSUSE instructions missing on "Distribution Packages" page
- Issue #3578[3119] - Build error when configuring with `HPX_HAVE_ALGORITHM_INPUT_ITERATOR_SUPPORT=ON`
- Issue #3575[3120] - Merge openSUSE reproducible patch
- Issue #3570[3121] - Update HPX to work with the latest VC version
- Issue #3567[3122] - Build succeed and make failed for `hpx:cout`
- Issue #3565[3123] - Polymorphic simple component destructor not getting called
- Issue #3559[3124] - 1.2.0 is missing from download page
- Issue #3554[3125] - Clang 6.0 warning of hiding overloaded virtual function
- Issue #3510[3126] - Build on ppc64 fails
- Issue #3482[3127] - Improve error message when `HPX_WITH_MAX_CPU_COUNT` is too low for given system

[3105] https://github.com/STEllAR-GROUP/hpx/issues/3641
[3106] https://github.com/STEllAR-GROUP/hpx/issues/3639
[3107] https://github.com/STEllAR-GROUP/hpx/issues/3634
[3108] https://github.com/STEllAR-GROUP/hpx/issues/3622
[3109] https://github.com/STEllAR-GROUP/hpx/issues/3616
[3110] https://github.com/STEllAR-GROUP/hpx/issues/3612
[3111] https://github.com/STEllAR-GROUP/hpx/issues/3609
[3112] https://github.com/STEllAR-GROUP/hpx/issues/3601
[3113] https://github.com/STEllAR-GROUP/hpx/issues/3598
[3114] https://github.com/STEllAR-GROUP/hpx/issues/3594
[3115] https://github.com/STEllAR-GROUP/hpx/issues/3592
[3116] https://github.com/STEllAR-GROUP/hpx/issues/3589
[3117] https://github.com/STEllAR-GROUP/hpx/issues/3587
[3118] https://github.com/STEllAR-GROUP/hpx/issues/3583
[3119] https://github.com/STEllAR-GROUP/hpx/issues/3578
[3120] https://github.com/STEllAR-GROUP/hpx/issues/3575
[3121] https://github.com/STEllAR-GROUP/hpx/issues/3570
[3122] https://github.com/STEllAR-GROUP/hpx/issues/3567
[3123] https://github.com/STEllAR-GROUP/hpx/issues/3565
[3124] https://github.com/STEllAR-GROUP/hpx/issues/3559
[3125] https://github.com/STEllAR-GROUP/hpx/issues/3554
[3126] https://github.com/STEllAR-GROUP/hpx/issues/3510
[3127] https://github.com/STEllAR-GROUP/hpx/issues/3482

- Issue #3453[3128] - Two HPX applications can't run at the same time.
- Issue #3452[3129] - Scaling issue on the change to 2 NUMA domains
- Issue #3442[3130] - HPX set_difference, set_intersection failure cases
- Issue #3437[3131] - Ensure parent_task pointer when child task is created and child/parent are on same locality
- Issue #3255[3132] - Suspension with lock for `--hpx:list-component-types`
- Issue #3034[3133] - Use C++17 structured bindings for serialization
- Issue #2999[3134] - Change thread scheduling use of `size_t` for thread indexing

Closed pull requests

- PR #3865[3135] - adds hpx_target_compile_option_if_available
- PR #3864[3136] - Helper functions that are useful in numa binding and testing of allocator
- PR #3862[3137] - Temporary fix to local_dataflow_boost_small_vector test
- PR #3860[3138] - Add cache line padding to intermediate results in for loop reduction
- PR #3859[3139] - Remove HPX_TLL_PUBLIC and HPX_TLL_PRIVATE from CMake files
- PR #3858[3140] - Add compile flags and definitions to modules
- PR #3851[3141] - update hpxmp release tag to v0.2.0
- PR #3849[3142] - Correct BOOST_ROOT variable name in quick start guide
- PR #3847[3143] - Fix attach_debugger configuration option
- PR #3846[3144] - Add tests for libs header tests
- PR #3844[3145] - Fixing source_groups in preprocessor module to properly handle compatibility headers
- PR #3843[3146] - This fixes the launch_process/launched_process pair of tests
- PR #3842[3147] - Fix macro call with ITTNOTIFY enabled
- PR #3840[3148] - Fixing SLURM environment parsing
- PR #3837[3149] - Fixing misplaced #endif

[3128] https://github.com/STEllAR-GROUP/hpx/issues/3453
[3129] https://github.com/STEllAR-GROUP/hpx/issues/3452
[3130] https://github.com/STEllAR-GROUP/hpx/issues/3442
[3131] https://github.com/STEllAR-GROUP/hpx/issues/3437
[3132] https://github.com/STEllAR-GROUP/hpx/issues/3255
[3133] https://github.com/STEllAR-GROUP/hpx/issues/3034
[3134] https://github.com/STEllAR-GROUP/hpx/issues/2999
[3135] https://github.com/STEllAR-GROUP/hpx/pull/3865
[3136] https://github.com/STEllAR-GROUP/hpx/pull/3864
[3137] https://github.com/STEllAR-GROUP/hpx/pull/3862
[3138] https://github.com/STEllAR-GROUP/hpx/pull/3860
[3139] https://github.com/STEllAR-GROUP/hpx/pull/3859
[3140] https://github.com/STEllAR-GROUP/hpx/pull/3858
[3141] https://github.com/STEllAR-GROUP/hpx/pull/3851
[3142] https://github.com/STEllAR-GROUP/hpx/pull/3849
[3143] https://github.com/STEllAR-GROUP/hpx/pull/3847
[3144] https://github.com/STEllAR-GROUP/hpx/pull/3846
[3145] https://github.com/STEllAR-GROUP/hpx/pull/3844
[3146] https://github.com/STEllAR-GROUP/hpx/pull/3843
[3147] https://github.com/STEllAR-GROUP/hpx/pull/3842
[3148] https://github.com/STEllAR-GROUP/hpx/pull/3840
[3149] https://github.com/STEllAR-GROUP/hpx/pull/3837

- PR #3835[3150] - make all latch members protected for consistency
- PR #3834[3151] - Disable transpose_block_numa example on CircleCI
- PR #3833[3152] - make latch **counter_** protected for deriving latch in hpxmp
- PR #3831[3153] - Fix CircleCI config for modules
- PR #3830[3154] - minor fix: option HPX_WITH_TEST was not working correctly
- PR #3828[3155] - Avoid for binaries that depend on HPX to directly link against internal modules
- PR #3827[3156] - Adding shortcut for `hpx::get_ptr<>(sync, id)` for a local, non-migratable objects
- PR #3826[3157] - Fix and update modules documentation
- PR #3825[3158] - Updating default APEX version to 2.1.3 with HPX
- PR #3823[3159] - Fix pkgconfig libs handling
- PR #3822[3160] - Change includes in hpx_wrap.cpp to more specific includes
- PR #3821[3161] - Disable barrier_3792 test when networking is disabled
- PR #3820[3162] - Assorted CMake fixes
- PR #3815[3163] - Removing left-over debug output
- PR #3814[3164] - Allow setting default scheduler mode via the configuration database
- PR #3813[3165] - Make the deprecation warnings issued by the old pp headers optional
- PR #3812[3166] - Windows requires to handle symlinks to directories differently from those linking files
- PR #3811[3167] - Clean up PP module and library skeleton
- PR #3806[3168] - Moving include path configuration to before APEX
- PR #3804[3169] - Fix latch
- PR #3803[3170] - Update hpxcxx to look at lib64 and use python3
- PR #3802[3171] - Numa binding allocator
- PR #3801[3172] - Remove duplicated includes

[3150] https://github.com/STEllAR-GROUP/hpx/pull/3835
[3151] https://github.com/STEllAR-GROUP/hpx/pull/3834
[3152] https://github.com/STEllAR-GROUP/hpx/pull/3833
[3153] https://github.com/STEllAR-GROUP/hpx/pull/3831
[3154] https://github.com/STEllAR-GROUP/hpx/pull/3830
[3155] https://github.com/STEllAR-GROUP/hpx/pull/3828
[3156] https://github.com/STEllAR-GROUP/hpx/pull/3827
[3157] https://github.com/STEllAR-GROUP/hpx/pull/3826
[3158] https://github.com/STEllAR-GROUP/hpx/pull/3825
[3159] https://github.com/STEllAR-GROUP/hpx/pull/3823
[3160] https://github.com/STEllAR-GROUP/hpx/pull/3822
[3161] https://github.com/STEllAR-GROUP/hpx/pull/3821
[3162] https://github.com/STEllAR-GROUP/hpx/pull/3820
[3163] https://github.com/STEllAR-GROUP/hpx/pull/3815
[3164] https://github.com/STEllAR-GROUP/hpx/pull/3814
[3165] https://github.com/STEllAR-GROUP/hpx/pull/3813
[3166] https://github.com/STEllAR-GROUP/hpx/pull/3812
[3167] https://github.com/STEllAR-GROUP/hpx/pull/3811
[3168] https://github.com/STEllAR-GROUP/hpx/pull/3806
[3169] https://github.com/STEllAR-GROUP/hpx/pull/3804
[3170] https://github.com/STEllAR-GROUP/hpx/pull/3803
[3171] https://github.com/STEllAR-GROUP/hpx/pull/3802
[3172] https://github.com/STEllAR-GROUP/hpx/pull/3801

- PR #3800[3173] - Attempt to fix Posix context switching after lazy init changes
- PR #3798[3174] - count and count_if accepts different iterator types
- PR #3797[3175] - Adding a couple of `override` keywords to overloaded virtual functions
- PR #3796[3176] - Re-enable testing all schedulers in shutdown_suspended_test
- PR #3795[3177] - Change `std::terminate` to std::abort in `SIGSEGV` handler
- PR #3794[3178] - Fixing #3792
- PR #3793[3179] - Extending migrate_polymorphic_component unit test
- PR #3791[3180] - Change `throw()` to `noexcept`
- PR #3790[3181] - Remove deprecated options for 1.3.0 release
- PR #3789[3182] - Remove Boost filesystem compatibility header
- PR #3788[3183] - Disabled even more spots that should not execute if networking is disabled
- PR #3787[3184] - Bump minimal boost supported version to 1.61.0
- PR #3786[3185] - Bump minimum required versions for 1.3.0 release
- PR #3785[3186] - Explicitly set number of jobs for all ninja invocations on CircleCI
- PR #3784[3187] - Fix leak and address sanitizer problems
- PR #3783[3188] - Disabled even more spots that should not execute is networking is disabled
- PR #3782[3189] - Cherry-picked tuple and thread_init_data fixes from #3701
- PR #3781[3190] - Fix generic context coroutines after lazy stack allocation changes
- PR #3780[3191] - Rename hello world examples
- PR #3776[3192] - Sort algorithms now use the supplied chunker to determine the required minimal chunk size
- PR #3775[3193] - Disable Boost auto-linking
- PR #3774[3194] - Tag and push stable builds
- PR #3773[3195] - Enable migration of polymorphic components

3173 https://github.com/STEllAR-GROUP/hpx/pull/3800
3174 https://github.com/STEllAR-GROUP/hpx/pull/3798
3175 https://github.com/STEllAR-GROUP/hpx/pull/3797
3176 https://github.com/STEllAR-GROUP/hpx/pull/3796
3177 https://github.com/STEllAR-GROUP/hpx/pull/3795
3178 https://github.com/STEllAR-GROUP/hpx/pull/3794
3179 https://github.com/STEllAR-GROUP/hpx/pull/3793
3180 https://github.com/STEllAR-GROUP/hpx/pull/3791
3181 https://github.com/STEllAR-GROUP/hpx/pull/3790
3182 https://github.com/STEllAR-GROUP/hpx/pull/3789
3183 https://github.com/STEllAR-GROUP/hpx/pull/3788
3184 https://github.com/STEllAR-GROUP/hpx/pull/3787
3185 https://github.com/STEllAR-GROUP/hpx/pull/3786
3186 https://github.com/STEllAR-GROUP/hpx/pull/3785
3187 https://github.com/STEllAR-GROUP/hpx/pull/3784
3188 https://github.com/STEllAR-GROUP/hpx/pull/3783
3189 https://github.com/STEllAR-GROUP/hpx/pull/3782
3190 https://github.com/STEllAR-GROUP/hpx/pull/3781
3191 https://github.com/STEllAR-GROUP/hpx/pull/3780
3192 https://github.com/STEllAR-GROUP/hpx/pull/3776
3193 https://github.com/STEllAR-GROUP/hpx/pull/3775
3194 https://github.com/STEllAR-GROUP/hpx/pull/3774
3195 https://github.com/STEllAR-GROUP/hpx/pull/3773

- PR #3771[3196] - Fix link to stackoverflow in documentation
- PR #3770[3197] - Replacing constexpr if in brace-serialization code
- PR #3769[3198] - Fix SIGSEGV handler
- PR #3768[3199] - Adding flags to scheduler allowing to control thread stealing and idle back-off
- PR #3767[3200] - Fix help formatting in hpxrun.py
- PR #3765[3201] - Fix a couple of bugs in the thread test
- PR #3764[3202] - Workaround for SFINAE regression in msvc14.2
- PR #3762[3203] - Prevent MSVC from prematurely instantiating things
- PR #3761[3204] - Update python scripts to work with python 3
- PR #3760[3205] - Fix callable vtable for GCC4.9
- PR #3759[3206] - Rename `PAGE_SIZE` to `PAGE_SIZE_` because AppleClang
- PR #3755[3207] - Making sure locks are not held during suspension
- PR #3754[3208] - Disable more code if networking is not available/not enabled
- PR #3752[3209] - Move `util::format` implementation to source file
- PR #3751[3210] - Fixing problems with `lcos::barrier` and iostreams
- PR #3750[3211] - Change error message to take into account `use_guard_page` setting
- PR #3749[3212] - Fix lifetime problem in `run_as_hpx_thread`
- PR #3748[3213] - Fixed unusable behavior of the clang code analyzer.
- PR #3747[3214] - Added `PMIX_RANK` to the defaults of `HPX_WITH_PARCELPORT_MPI_ENV`.
- PR #3745[3215] - Introduced `cache_aligned_data` and `cache_line_data` helper structure
- PR #3742[3216] - Remove more unused functionality from util/logging
- PR #3740[3217] - Fix includes in partitioned vector tests
- PR #3739[3218] - More fixes to make sure that `std::flush` really flushes all output

[3196] https://github.com/STEllAR-GROUP/hpx/pull/3771
[3197] https://github.com/STEllAR-GROUP/hpx/pull/3770
[3198] https://github.com/STEllAR-GROUP/hpx/pull/3769
[3199] https://github.com/STEllAR-GROUP/hpx/pull/3768
[3200] https://github.com/STEllAR-GROUP/hpx/pull/3767
[3201] https://github.com/STEllAR-GROUP/hpx/pull/3765
[3202] https://github.com/STEllAR-GROUP/hpx/pull/3764
[3203] https://github.com/STEllAR-GROUP/hpx/pull/3762
[3204] https://github.com/STEllAR-GROUP/hpx/pull/3761
[3205] https://github.com/STEllAR-GROUP/hpx/pull/3760
[3206] https://github.com/STEllAR-GROUP/hpx/pull/3759
[3207] https://github.com/STEllAR-GROUP/hpx/pull/3755
[3208] https://github.com/STEllAR-GROUP/hpx/pull/3754
[3209] https://github.com/STEllAR-GROUP/hpx/pull/3752
[3210] https://github.com/STEllAR-GROUP/hpx/pull/3751
[3211] https://github.com/STEllAR-GROUP/hpx/pull/3750
[3212] https://github.com/STEllAR-GROUP/hpx/pull/3749
[3213] https://github.com/STEllAR-GROUP/hpx/pull/3748
[3214] https://github.com/STEllAR-GROUP/hpx/pull/3747
[3215] https://github.com/STEllAR-GROUP/hpx/pull/3745
[3216] https://github.com/STEllAR-GROUP/hpx/pull/3742
[3217] https://github.com/STEllAR-GROUP/hpx/pull/3740
[3218] https://github.com/STEllAR-GROUP/hpx/pull/3739

- PR #3737[3219] - Fix potential shutdown problems
- PR #3736[3220] - Fix `guided_pool_executor` after dataflow changes caused compilation fail
- PR #3734[3221] - Limiting executor
- PR #3732[3222] - More constrained bound constructors
- PR #3730[3223] - Attempt to fix deadlocks during component loading
- PR #3729[3224] - Add latch member function `count_up` and reset, requested by hpxMP
- PR #3728[3225] - Send even empty buffers on `hpx::endl` and `hpx::flush`
- PR #3727[3226] - Adding example demonstrating how to customize the memory management for a component
- PR #3726[3227] - Adding support for passing command line options through the `HPX_COMMANDLINE_OPTIONS` environment variable
- PR #3722[3228] - Document known broken OpenMPI builds
- PR #3716[3229] - Add barrier reset function, requested by hpxMP for reusing barrier
- PR #3715[3230] - More work on functions and vtables
- PR #3714[3231] - Generate single-page HTML, PDF, manpage from documentation
- PR #3713[3232] - Updating default APEX version to 2.1.2
- PR #3712[3233] - Update release procedure
- PR #3710[3234] - Fix the C++11 build, after #3704
- PR #3709[3235] - Move some component_registry functionality to source file
- PR #3708[3236] - Ignore all locks while handling assertions
- PR #3707[3237] - Remove obsolete hpx runtime executable
- PR #3705[3238] - Fix and simplify `make_ready_future` overload sets
- PR #3704[3239] - Reduce use of binders
- PR #3703[3240] - Ini
- PR #3702[3241] - Fixing CUDA compiler errors

[3219] https://github.com/STEllAR-GROUP/hpx/pull/3737
[3220] https://github.com/STEllAR-GROUP/hpx/pull/3736
[3221] https://github.com/STEllAR-GROUP/hpx/pull/3734
[3222] https://github.com/STEllAR-GROUP/hpx/pull/3732
[3223] https://github.com/STEllAR-GROUP/hpx/pull/3730
[3224] https://github.com/STEllAR-GROUP/hpx/pull/3729
[3225] https://github.com/STEllAR-GROUP/hpx/pull/3728
[3226] https://github.com/STEllAR-GROUP/hpx/pull/3727
[3227] https://github.com/STEllAR-GROUP/hpx/pull/3726
[3228] https://github.com/STEllAR-GROUP/hpx/pull/3722
[3229] https://github.com/STEllAR-GROUP/hpx/pull/3716
[3230] https://github.com/STEllAR-GROUP/hpx/pull/3715
[3231] https://github.com/STEllAR-GROUP/hpx/pull/3714
[3232] https://github.com/STEllAR-GROUP/hpx/pull/3713
[3233] https://github.com/STEllAR-GROUP/hpx/pull/3712
[3234] https://github.com/STEllAR-GROUP/hpx/pull/3710
[3235] https://github.com/STEllAR-GROUP/hpx/pull/3709
[3236] https://github.com/STEllAR-GROUP/hpx/pull/3708
[3237] https://github.com/STEllAR-GROUP/hpx/pull/3707
[3238] https://github.com/STEllAR-GROUP/hpx/pull/3705
[3239] https://github.com/STEllAR-GROUP/hpx/pull/3704
[3240] https://github.com/STEllAR-GROUP/hpx/pull/3703
[3241] https://github.com/STEllAR-GROUP/hpx/pull/3702

- PR #3700[3242] - Added `barrier::increment` function to increase total number of thread
- PR #3697[3243] - One more attempt to fix migration…
- PR #3694[3244] - Fixing component migration
- PR #3693[3245] - Print thread state when getting disallowed value in set_thread_state
- PR #3692[3246] - Only disable `constexpr` with clang-cuda, not nvcc+gcc
- PR #3691[3247] - Link with libsupc++ if needed for thread_local
- PR #3690[3248] - Remove thousands separators in set_operations_3442 to comply with C++11
- PR #3688[3249] - Decouple serialization from function vtables
- PR #3687[3250] - Fix a couple of test failures
- PR #3686[3251] - Make sure tests.unit.build are run after install on CircleCI
- PR #3685[3252] - Revise quickstart CMakeLists.txt explanation
- PR #3684[3253] - Provide concept emulation for Ranges-TS concepts
- PR #3683[3254] - Ignore uninitialized chunks
- PR #3682[3255] - Ignore uninitialized chunks. Check proper indices.
- PR #3680[3256] - Ignore uninitialized chunks. Check proper range indices
- PR #3679[3257] - Simplify basic action implementations
- PR #3678[3258] - Making sure `HPX_HAVE_LIBATOMIC` is unset before checking
- PR #3677[3259] - Fix generated full version number to be usable in expressions
- PR #3674[3260] - Reduce functional utilities call depth
- PR #3672[3261] - Change new build system to use existing macros related to pseudo dependencies
- PR #3669[3262] - Remove indirection in `function_ref` when thread description is disabled
- PR #3668[3263] - Unbreaking `async_*cb*` tests
- PR #3667[3264] - Generate version.hpp

3242 https://github.com/STEllAR-GROUP/hpx/pull/3700
3243 https://github.com/STEllAR-GROUP/hpx/pull/3697
3244 https://github.com/STEllAR-GROUP/hpx/pull/3694
3245 https://github.com/STEllAR-GROUP/hpx/pull/3693
3246 https://github.com/STEllAR-GROUP/hpx/pull/3692
3247 https://github.com/STEllAR-GROUP/hpx/pull/3691
3248 https://github.com/STEllAR-GROUP/hpx/pull/3690
3249 https://github.com/STEllAR-GROUP/hpx/pull/3688
3250 https://github.com/STEllAR-GROUP/hpx/pull/3687
3251 https://github.com/STEllAR-GROUP/hpx/pull/3686
3252 https://github.com/STEllAR-GROUP/hpx/pull/3685
3253 https://github.com/STEllAR-GROUP/hpx/pull/3684
3254 https://github.com/STEllAR-GROUP/hpx/pull/3683
3255 https://github.com/STEllAR-GROUP/hpx/pull/3682
3256 https://github.com/STEllAR-GROUP/hpx/pull/3680
3257 https://github.com/STEllAR-GROUP/hpx/pull/3679
3258 https://github.com/STEllAR-GROUP/hpx/pull/3678
3259 https://github.com/STEllAR-GROUP/hpx/pull/3677
3260 https://github.com/STEllAR-GROUP/hpx/pull/3674
3261 https://github.com/STEllAR-GROUP/hpx/pull/3672
3262 https://github.com/STEllAR-GROUP/hpx/pull/3669
3263 https://github.com/STEllAR-GROUP/hpx/pull/3668
3264 https://github.com/STEllAR-GROUP/hpx/pull/3667

- PR #3665[3265] - Enabling MPI parcelport for gitlab runners
- PR #3664[3266] - making clang-tidy work properly again
- PR #3662[3267] - Attempt to fix exception handling
- PR #3661[3268] - Move `lcos::latch` to source file
- PR #3660[3269] - Fix accidentally explicit gid_type default constructor
- PR #3659[3270] - Parallel executor latch
- PR #3658[3271] - Fixing execution_parameters
- PR #3657[3272] - Avoid dangling references in wait_all
- PR #3656[3273] - Avoiding lifetime problems with sync_put_parcel
- PR #3655[3274] - Fixing nullptr dereference inside of function
- PR #3652[3275] - Attempt to fix `thread_map_type` definition with C++11
- PR #3650[3276] - Allowing for end iterator being different from begin iterator
- PR #3649[3277] - Added architecture identification to cmake to be able to detect timestamp support
- PR #3645[3278] - Enabling sanitizers on gitlab runner
- PR #3644[3279] - Attempt to tackle timeouts during startup
- PR #3642[3280] - Cleanup parallel partitioners
- PR #3640[3281] - Dataflow now works with functions that return a reference
- PR #3637[3282] - Merging the executor-enabled overloads of `shared_future<>::then`
- PR #3633[3283] - Replace deprecated boost endian macros
- PR #3632[3284] - Add instructions on getting HPX to documentation
- PR #3631[3285] - Simplify parcel creation
- PR #3630[3286] - Small additions and fixes to release procedure
- PR #3629[3287] - Modular pp

3265 https://github.com/STEllAR-GROUP/hpx/pull/3665
3266 https://github.com/STEllAR-GROUP/hpx/pull/3664
3267 https://github.com/STEllAR-GROUP/hpx/pull/3662
3268 https://github.com/STEllAR-GROUP/hpx/pull/3661
3269 https://github.com/STEllAR-GROUP/hpx/pull/3660
3270 https://github.com/STEllAR-GROUP/hpx/pull/3659
3271 https://github.com/STEllAR-GROUP/hpx/pull/3658
3272 https://github.com/STEllAR-GROUP/hpx/pull/3657
3273 https://github.com/STEllAR-GROUP/hpx/pull/3656
3274 https://github.com/STEllAR-GROUP/hpx/pull/3655
3275 https://github.com/STEllAR-GROUP/hpx/pull/3652
3276 https://github.com/STEllAR-GROUP/hpx/pull/3650
3277 https://github.com/STEllAR-GROUP/hpx/pull/3649
3278 https://github.com/STEllAR-GROUP/hpx/pull/3645
3279 https://github.com/STEllAR-GROUP/hpx/pull/3644
3280 https://github.com/STEllAR-GROUP/hpx/pull/3642
3281 https://github.com/STEllAR-GROUP/hpx/pull/3640
3282 https://github.com/STEllAR-GROUP/hpx/pull/3637
3283 https://github.com/STEllAR-GROUP/hpx/pull/3633
3284 https://github.com/STEllAR-GROUP/hpx/pull/3632
3285 https://github.com/STEllAR-GROUP/hpx/pull/3631
3286 https://github.com/STEllAR-GROUP/hpx/pull/3630
3287 https://github.com/STEllAR-GROUP/hpx/pull/3629

- PR #3627[3288] - Implement `util::function_ref`
- PR #3626[3289] - Fix cancelable_action_client example
- PR #3625[3290] - Added automatic serialization for simple structs (see #3034)
- PR #3624[3291] - Updating the default order of priority for `thread_description`
- PR #3621[3292] - Update copyright year and other small formatting fixes
- PR #3620[3293] - Adding support for gitlab runner
- PR #3619[3294] - Store debug logs and core dumps on CircleCI
- PR #3618[3295] - Various optimizations
- PR #3617[3296] - Fix link to the gpg key (#2)
- PR #3615[3297] - Fix unused variable warnings with networking off
- PR #3614[3298] - Restructuring counter data in scheduler to reduce false sharing
- PR #3613[3299] - Adding support for gitlab runners
- PR #3610[3300] - Don't wait for `stop_condition` in main thread
- PR #3608[3301] - Add inline keyword to `invalid_thread_id` definition for nvcc
- PR #3607[3302] - Adding configuration key that allows one to explicitly add a directory to the component search path
- PR #3606[3303] - Add nvcc to exclude constexpress since is it not supported by nvcc
- PR #3605[3304] - Add `inline` to definition of checkpoint stream operators to fix link error
- PR #3604[3305] - Use format for string formatting
- PR #3603[3306] - Improve the error message for using to less `MAX_CPU_COUNT`
- PR #3602[3307] - Improve the error message for to small values of `MAX_CPU_COUNT`
- PR #3600[3308] - Parallel executor aggregated
- PR #3599[3309] - Making sure networking is disabled for default one-locality-runs
- PR #3596[3310] - Store thread exit functions in `forward_list` instead of `deque` to avoid allocations

[3288] https://github.com/STEllAR-GROUP/hpx/pull/3627
[3289] https://github.com/STEllAR-GROUP/hpx/pull/3626
[3290] https://github.com/STEllAR-GROUP/hpx/pull/3625
[3291] https://github.com/STEllAR-GROUP/hpx/pull/3624
[3292] https://github.com/STEllAR-GROUP/hpx/pull/3621
[3293] https://github.com/STEllAR-GROUP/hpx/pull/3620
[3294] https://github.com/STEllAR-GROUP/hpx/pull/3619
[3295] https://github.com/STEllAR-GROUP/hpx/pull/3618
[3296] https://github.com/STEllAR-GROUP/hpx/pull/3617
[3297] https://github.com/STEllAR-GROUP/hpx/pull/3615
[3298] https://github.com/STEllAR-GROUP/hpx/pull/3614
[3299] https://github.com/STEllAR-GROUP/hpx/pull/3613
[3300] https://github.com/STEllAR-GROUP/hpx/pull/3610
[3301] https://github.com/STEllAR-GROUP/hpx/pull/3608
[3302] https://github.com/STEllAR-GROUP/hpx/pull/3607
[3303] https://github.com/STEllAR-GROUP/hpx/pull/3606
[3304] https://github.com/STEllAR-GROUP/hpx/pull/3605
[3305] https://github.com/STEllAR-GROUP/hpx/pull/3604
[3306] https://github.com/STEllAR-GROUP/hpx/pull/3603
[3307] https://github.com/STEllAR-GROUP/hpx/pull/3602
[3308] https://github.com/STEllAR-GROUP/hpx/pull/3600
[3309] https://github.com/STEllAR-GROUP/hpx/pull/3599
[3310] https://github.com/STEllAR-GROUP/hpx/pull/3596

- PR #3590[3311] - Fix typo/mistake in thread queue `cleanup_terminated`
- PR #3588[3312] - Fix formatting errors in launching_and_configuring_hpx_applications.rst
- PR #3586[3313] - Make bind propagate value category
- PR #3585[3314] - Extend Cmake for building hpx as distribution packages (refs #3575)
- PR #3584[3315] - Untangle function storage from object pointer
- PR #3582[3316] - Towards Modularized HPX
- PR #3580[3317] - Remove extra `||` in merge.hpp
- PR #3577[3318] - Partially revert "Remove vtable empty flag"
- PR #3576[3319] - Make sure empty startup/shutdown functions are not being used
- PR #3574[3320] - Make sure `DATAPAR` settings are conveyed to depending projects
- PR #3573[3321] - Make sure HPX is usable with latest released version of Vc (V1.4.1)
- PR #3572[3322] - Adding test ensuring ticket 3565 is fixed
- PR #3571[3323] - Make empty `[unique_]function` vtable non-dependent
- PR #3566[3324] - Fix compilation with dynamic bitset for CPU masks
- PR #3563[3325] - Drop `util::[unique_]function` target_type
- PR #3562[3326] - Removing the target suffixes
- PR #3561[3327] - Replace executor traits return type deduction (keep non-SFINAE)
- PR #3557[3328] - Replace the last usages of boost::atomic
- PR #3556[3329] - Replace `boost::scoped_array` with `std::unique_ptr`
- PR #3552[3330] - (Re)move APEX readme
- PR #3548[3331] - Replace `boost::scoped_ptr` with `std::unique_ptr`
- PR #3547[3332] - Remove last use of Boost.Signals2
- PR #3544[3333] - Post 1.2.0 version bumps

[3311] https://github.com/STEllAR-GROUP/hpx/pull/3590
[3312] https://github.com/STEllAR-GROUP/hpx/pull/3588
[3313] https://github.com/STEllAR-GROUP/hpx/pull/3586
[3314] https://github.com/STEllAR-GROUP/hpx/pull/3585
[3315] https://github.com/STEllAR-GROUP/hpx/pull/3584
[3316] https://github.com/STEllAR-GROUP/hpx/pull/3582
[3317] https://github.com/STEllAR-GROUP/hpx/pull/3580
[3318] https://github.com/STEllAR-GROUP/hpx/pull/3577
[3319] https://github.com/STEllAR-GROUP/hpx/pull/3576
[3320] https://github.com/STEllAR-GROUP/hpx/pull/3574
[3321] https://github.com/STEllAR-GROUP/hpx/pull/3573
[3322] https://github.com/STEllAR-GROUP/hpx/pull/3572
[3323] https://github.com/STEllAR-GROUP/hpx/pull/3571
[3324] https://github.com/STEllAR-GROUP/hpx/pull/3566
[3325] https://github.com/STEllAR-GROUP/hpx/pull/3563
[3326] https://github.com/STEllAR-GROUP/hpx/pull/3562
[3327] https://github.com/STEllAR-GROUP/hpx/pull/3561
[3328] https://github.com/STEllAR-GROUP/hpx/pull/3557
[3329] https://github.com/STEllAR-GROUP/hpx/pull/3556
[3330] https://github.com/STEllAR-GROUP/hpx/pull/3552
[3331] https://github.com/STEllAR-GROUP/hpx/pull/3548
[3332] https://github.com/STEllAR-GROUP/hpx/pull/3547
[3333] https://github.com/STEllAR-GROUP/hpx/pull/3544

- PR #3543[3334] - added Ubuntu dependency list to readme
- PR #3531[3335] - Warnings, warnings…
- PR #3527[3336] - Add CircleCI filter for building all tags
- PR #3525[3337] - Segmented algorithms
- PR #3517[3338] - Replace `boost::regex` with C++11 `<regex>`
- PR #3514[3339] - Cleaning up the build system
- PR #3505[3340] - Fixing type attribute warning for `transfer_action`
- PR #3504[3341] - Add support for rpm packaging
- PR #3499[3342] - Improving spinlock pools
- PR #3498[3343] - Remove thread specific ptr
- PR #3486[3344] - Fix comparison for expect_connecting_localities config entry
- PR #3469[3345] - Enable (existing) code for extracting stack pointer on Power platform

HPX V1.2.1 (Feb 19, 2019)

General changes

This is a bugfix release. It contains the following changes:

- Fix compilation on ARM, s390x and 32-bit architectures.
- Fix a critical bug in the `future` implementation.
- Fix several problems in the CMake configuration which affects external projects.
- Add support for Boost 1.69.0.

Closed issues

- Issue #3638[3346] - Build HPX 1.2 with boost 1.69
- Issue #3635[3347] - Non-deterministic crashing on Stampede2
- Issue #3550[3348] - 1>e:000workhpxsrcthrow_exception.cpp(54): error C2440: '<function-style-cast>': cannot convert from 'boost::system::error_code' to 'hpx::exception'

[3334] https://github.com/STEllAR-GROUP/hpx/pull/3543
[3335] https://github.com/STEllAR-GROUP/hpx/pull/3531
[3336] https://github.com/STEllAR-GROUP/hpx/pull/3527
[3337] https://github.com/STEllAR-GROUP/hpx/pull/3525
[3338] https://github.com/STEllAR-GROUP/hpx/pull/3517
[3339] https://github.com/STEllAR-GROUP/hpx/pull/3514
[3340] https://github.com/STEllAR-GROUP/hpx/pull/3505
[3341] https://github.com/STEllAR-GROUP/hpx/pull/3504
[3342] https://github.com/STEllAR-GROUP/hpx/pull/3499
[3343] https://github.com/STEllAR-GROUP/hpx/pull/3498
[3344] https://github.com/STEllAR-GROUP/hpx/pull/3486
[3345] https://github.com/STEllAR-GROUP/hpx/pull/3469
[3346] https://github.com/STEllAR-GROUP/hpx/issues/3638
[3347] https://github.com/STEllAR-GROUP/hpx/issues/3635
[3348] https://github.com/STEllAR-GROUP/hpx/issues/3550

- Issue #3549[3349] - HPX 1.2.0 does not build on i686, but release candidate did
- Issue #3511[3350] - Build on s390x fails
- Issue #3509[3351] - Build on armv7l fails

Closed pull requests

- PR #3695[3352] - Don't install CMake templates and packaging files
- PR #3666[3353] - Fixing yet another race in future_data
- PR #3663[3354] - Fixing race between setting and getting the value inside future_data
- PR #3648[3355] - Adding timestamp option for S390x platform
- PR #3647[3356] - Blind attempt to fix warnings issued by gcc V9
- PR #3611[3357] - Include GNUInstallDirs earlier to have it available for subdirectories
- PR #3595[3358] - Use GNUInstallDirs lib path in pkgconfig config file
- PR #3593[3359] - Add include(GNUInstallDirs) to HPXMacros.cmake
- PR #3591[3360] - Fix compilation error on arm7 architecture. Compiles and runs on Fedora 29 on Pi 3.
- PR #3558[3361] - Adding constructor *exception(boost::system::error_code const&)*
- PR #3555[3362] - cmake: make install locations configurable
- PR #3551[3363] - Fix uint64_t causing compilation fail on i686

HPX V1.2.0 (Nov 12, 2018)

General changes

Here are some of the main highlights and changes for this release:

- Thanks to the work of our Google Summer of Code student, Nikunj Gupta, we now have a new implementation of `hpx_main.hpp` on supported platforms (Linux, BSD and MacOS). This is intended to be a less fragile drop-in replacement for the old implementation relying on preprocessor macros. The new implementation does not require changes if you are using the CMake[3364] or pkg-config. The old behaviour can be restored by setting `HPX_WITH_DYNAMIC_HPX_MAIN=OFF` during CMake[3365] configuration. The implementation on Windows is unchanged.

3349 https://github.com/STEllAR-GROUP/hpx/issues/3549
3350 https://github.com/STEllAR-GROUP/hpx/issues/3511
3351 https://github.com/STEllAR-GROUP/hpx/issues/3509
3352 https://github.com/STEllAR-GROUP/hpx/pull/3695
3353 https://github.com/STEllAR-GROUP/hpx/pull/3666
3354 https://github.com/STEllAR-GROUP/hpx/pull/3663
3355 https://github.com/STEllAR-GROUP/hpx/pull/3648
3356 https://github.com/STEllAR-GROUP/hpx/pull/3647
3357 https://github.com/STEllAR-GROUP/hpx/pull/3611
3358 https://github.com/STEllAR-GROUP/hpx/pull/3595
3359 https://github.com/STEllAR-GROUP/hpx/pull/3593
3360 https://github.com/STEllAR-GROUP/hpx/pull/3591
3361 https://github.com/STEllAR-GROUP/hpx/pull/3558
3362 https://github.com/STEllAR-GROUP/hpx/pull/3555
3363 https://github.com/STEllAR-GROUP/hpx/pull/3551
3364 https://www.cmake.org
3365 https://www.cmake.org

- We have added functionality to allow passing scheduling hints to our schedulers. These will allow us to create executors that for example target a specific NUMA domain or allow for *HPX* threads to be pinned to a particular worker thread.
- We have significantly improved the performance of our futures implementation by making the shared state atomic.
- We have replaced Boostbook by Sphinx for our documentation. This means the documentation is easier to navigate with built-in search and table of contents. We have also added a quick start section and restructured the documentation to be easier to follow for new users.
- We have added a new option to the *--hpx:threads* command line option. It is now possible to use `cores` to tell *HPX* to only use one worker thread per core, unlike the existing option `all` which uses one worker thread per processing unit (processing unit can be a hyperthread if hyperthreads are available). The default value of *--hpx:threads* has also been changed to `cores` as this leads to better performance in most cases.
- All command line options can now be passed alongside configuration options when initializing *HPX*. This means that some options that were previously only available on the command line can now be set as configuration options.
- HPXMP is a portable, scalable, and flexible application programming interface using the OpenMP specification that supports multi-platform shared memory multiprocessing programming in C and C++. HPXMP can be enabled within *HPX* by setting `DHPX_WITH_HPXMP=ON` during CMake[3366] configuration.
- Two new performance counters were added for measuring the time spent doing background work. `/threads/time/background-work-duration` returns the time spent doing background on a given thread or locality, while `/threads/time/background-overhead` returns the fraction of time spent doing background work with respect to the overall time spent running the scheduler. The new performance counters are disabled by default and can be turned on by setting `HPX_WITH_BACKGROUND_THREAD_COUNTERS=ON` during CMake[3367] configuration.
- The idling behaviour of *HPX* has been tweaked to allow for faster idling. This is useful in interactive applications where the *HPX* worker threads may not have work all the time. This behaviour can be tweaked and turned off as before with `HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF` during CMake[3368] configuration.
- It is now possible to register callback functions for *HPX* worker thread events. Callbacks can be registered for starting and stopping worker threads, and for when errors occur.

Breaking changes

- The implementation of `hpx_main.hpp` has changed. If you are using custom Makefiles you will need to make changes. Please see the documentation on *using Makefiles* for more details.
- The default value of *--hpx:threads* has changed from `all` to `cores`. The new option `cores` only starts one worker thread per core.
- We have dropped support for Boost 1.56 and 1.57. The minimal version of Boost we now test is 1.58.
- Our `boost::format`-based formatting implementation has been revised and replaced with a custom implementation. This changes the formatting syntax and requires changes if you are relying on `hpx::util::format` or `hpx::util::format_to`. The pull request for this change contains more information: PR #3266[3369].
- The following deprecated options have now been completely removed: `HPX_WITH_ASYNC_FUNCTION_COMPATIBILITY`, `HPX_WITH_LOCAL_DATAFLOW`, `HPX_WITH_GENERIC_EXECUTION_POLICY`, `HPX_WITH_BOOST_CHRONO_COMPATIBILITY`, `HPX_WITH_EXECUTOR_COMPATIBILITY`, `HPX_WITH_EXECUTION_POLICY_COMPATIBILITY`, and `HPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY`.

[3366] https://www.cmake.org
[3367] https://www.cmake.org
[3368] https://www.cmake.org
[3369] https://github.com/STEllAR-GROUP/hpx/pull/3266

Closed issues

- Issue #3538[3370] - numa handling incorrect for hwloc 2
- Issue #3533[3371] - Cmake version 3.5.1does not work (git ff26b35 2018-11-06)
- Issue #3526[3372] - Failed building hpx-1.2.0-rc1 on Ubuntu16.04 x86-64 Virtualbox VM
- Issue #3512[3373] - Build on aarch64 fails
- Issue #3475[3374] - HPX fails to link if the MPI parcelport is enabled
- Issue #3462[3375] - CMake configuration shows a minor and inconsequential failure to create a symlink
- Issue #3461[3376] - Compilation Problems with the most recent Clang
- Issue #3460[3377] - Deadlock when create_partitioner fails (assertion fails) in debug mode
- Issue #3455[3378] - HPX build failing with HWLOC errors on POWER8 with hwloc 1.8
- Issue #3438[3379] - HPX no longer builds on IBM POWER8
- Issue #3426[3380] - hpx build failed on MacOS
- Issue #3424[3381] - CircleCI builds broken for forked repositories
- Issue #3422[3382] - Benchmarks in tests.performance.local are not run nightly
- Issue #3408[3383] - CMake Targets for HPX
- Issue #3399[3384] - processing unit out of bounds
- Issue #3395[3385] - Floating point bug in hpx/runtime/threads/policies/scheduler_base.hpp
- Issue #3378[3386] - compile error with lcos::communicator
- Issue #3376[3387] - Failed to build HPX with APEX using clang
- Issue #3366[3388] - Adapted Safe_Object example fails for –hpx:threads > 1
- Issue #3360[3389] - Segmentation fault when passing component id as parameter
- Issue #3358[3390] - HPX runtime hangs after multiple (~thousands) start-stop sequences
- Issue #3352[3391] - Support TCP provider in libfabric ParcelPort

[3370] https://github.com/STEllAR-GROUP/hpx/issues/3538
[3371] https://github.com/STEllAR-GROUP/hpx/issues/3533
[3372] https://github.com/STEllAR-GROUP/hpx/issues/3526
[3373] https://github.com/STEllAR-GROUP/hpx/issues/3512
[3374] https://github.com/STEllAR-GROUP/hpx/issues/3475
[3375] https://github.com/STEllAR-GROUP/hpx/issues/3462
[3376] https://github.com/STEllAR-GROUP/hpx/issues/3461
[3377] https://github.com/STEllAR-GROUP/hpx/issues/3460
[3378] https://github.com/STEllAR-GROUP/hpx/issues/3455
[3379] https://github.com/STEllAR-GROUP/hpx/issues/3438
[3380] https://github.com/STEllAR-GROUP/hpx/issues/3426
[3381] https://github.com/STEllAR-GROUP/hpx/issues/3424
[3382] https://github.com/STEllAR-GROUP/hpx/issues/3422
[3383] https://github.com/STEllAR-GROUP/hpx/issues/3408
[3384] https://github.com/STEllAR-GROUP/hpx/issues/3399
[3385] https://github.com/STEllAR-GROUP/hpx/issues/3395
[3386] https://github.com/STEllAR-GROUP/hpx/issues/3378
[3387] https://github.com/STEllAR-GROUP/hpx/issues/3376
[3388] https://github.com/STEllAR-GROUP/hpx/issues/3366
[3389] https://github.com/STEllAR-GROUP/hpx/issues/3360
[3390] https://github.com/STEllAR-GROUP/hpx/issues/3358
[3391] https://github.com/STEllAR-GROUP/hpx/issues/3352

- Issue #3342[3392] - undefined reference to __atomic_load_16
- Issue #3339[3393] - setting command line options/flags from init cfg is not obvious
- Issue #3325[3394] - AGAS migrates components prematurely
- Issue #3321[3395] - hpx bad_parameter handling is awful
- Issue #3318[3396] - Benchmarks fail to build with C++11
- Issue #3304[3397] - hpx::threads::run_as_hpx_thread does not properly handle exceptions
- Issue #3300[3398] - Setting pu step or offset results in no threads in default pool
- Issue #3297[3399] - Crash with APEX when running Phylanx lra_csv with > 1 thread
- Issue #3296[3400] - Building HPX with APEX configuration gives compiler warnings
- Issue #3290[3401] - make tests failing at hello_world_component
- Issue #3285[3402] - possible compilation error when "using namespace std;" is defined before including "hpx" headers files
- Issue #3280[3403] - HPX fails on OSX
- Issue #3272[3404] - CircleCI does not upload generated docker image any more
- Issue #3270[3405] - Error when compiling CUDA examples
- Issue #3267[3406] - `tests.unit.host_.block_allocator` fails occasionally
- Issue #3264[3407] - Possible move to Sphinx for documentation
- Issue #3263[3408] - Documentation improvements
- Issue #3259[3409] - `set_parcel_write_handler` test fails occasionally
- Issue #3258[3410] - Links to source code in documentation are broken
- Issue #3247[3411] - Rare `tests.unit.host_.block_allocator` test failure on 1.1.0-rc1
- Issue #3244[3412] - Slowing down and speeding up an interval_timer
- Issue #3215[3413] - Cannot build both tests and examples on MSVC with pseudo-dependencies enabled
- Issue #3195[3414] - Unnecessary customization point route causing performance penalty

3392 https://github.com/STEllAR-GROUP/hpx/issues/3342
3393 https://github.com/STEllAR-GROUP/hpx/issues/3339
3394 https://github.com/STEllAR-GROUP/hpx/issues/3325
3395 https://github.com/STEllAR-GROUP/hpx/issues/3321
3396 https://github.com/STEllAR-GROUP/hpx/issues/3318
3397 https://github.com/STEllAR-GROUP/hpx/issues/3304
3398 https://github.com/STEllAR-GROUP/hpx/issues/3300
3399 https://github.com/STEllAR-GROUP/hpx/issues/3297
3400 https://github.com/STEllAR-GROUP/hpx/issues/3296
3401 https://github.com/STEllAR-GROUP/hpx/issues/3290
3402 https://github.com/STEllAR-GROUP/hpx/issues/3285
3403 https://github.com/STEllAR-GROUP/hpx/issues/3280
3404 https://github.com/STEllAR-GROUP/hpx/issues/3272
3405 https://github.com/STEllAR-GROUP/hpx/issues/3270
3406 https://github.com/STEllAR-GROUP/hpx/issues/3267
3407 https://github.com/STEllAR-GROUP/hpx/issues/3264
3408 https://github.com/STEllAR-GROUP/hpx/issues/3263
3409 https://github.com/STEllAR-GROUP/hpx/issues/3259
3410 https://github.com/STEllAR-GROUP/hpx/issues/3258
3411 https://github.com/STEllAR-GROUP/hpx/issues/3247
3412 https://github.com/STEllAR-GROUP/hpx/issues/3244
3413 https://github.com/STEllAR-GROUP/hpx/issues/3215
3414 https://github.com/STEllAR-GROUP/hpx/issues/3195

- Issue #3088[3415] - A strange thing in parallel::sort.
- Issue #2650[3416] - libfabric support for passive endpoints
- Issue #1205[3417] - TSS is broken

Closed pull requests

- PR #3542[3418] - Fix numa lookup from pu when using hwloc 2.x
- PR #3541[3419] - Fixing the build system of the MPI parcelport
- PR #3540[3420] - Updating HPX people section
- PR #3539[3421] - Splitting test to avoid OOM on CircleCI
- PR #3537[3422] - Fix guided exec
- PR #3536[3423] - Updating grants which support the LSU team
- PR #3535[3424] - Fix hiding of docker credentials
- PR #3534[3425] - Fixing #3533
- PR #3532[3426] - fixing minor doc typo –hpx:print-counter-at arg
- PR #3530[3427] - Changing APEX default tag to v2.1.0
- PR #3529[3428] - Remove leftover security options and documentation
- PR #3528[3429] - Fix hwloc version check
- PR #3524[3430] - Do not build guided pool examples with older GCC compilers
- PR #3523[3431] - Fix logging regression
- PR #3522[3432] - Fix more warnings
- PR #3521[3433] - Fixing argument handling in induction and reduction clauses for parallel::for_loop
- PR #3520[3434] - Remove docs symlink and versioned docs folders
- PR #3519[3435] - hpxMP release
- PR #3518[3436] - Change all steps to use new docker image on CircleCI

[3415] https://github.com/STEllAR-GROUP/hpx/issues/3088
[3416] https://github.com/STEllAR-GROUP/hpx/issues/2650
[3417] https://github.com/STEllAR-GROUP/hpx/issues/1205
[3418] https://github.com/STEllAR-GROUP/hpx/pull/3542
[3419] https://github.com/STEllAR-GROUP/hpx/pull/3541
[3420] https://github.com/STEllAR-GROUP/hpx/pull/3540
[3421] https://github.com/STEllAR-GROUP/hpx/pull/3539
[3422] https://github.com/STEllAR-GROUP/hpx/pull/3537
[3423] https://github.com/STEllAR-GROUP/hpx/pull/3536
[3424] https://github.com/STEllAR-GROUP/hpx/pull/3535
[3425] https://github.com/STEllAR-GROUP/hpx/pull/3534
[3426] https://github.com/STEllAR-GROUP/hpx/pull/3532
[3427] https://github.com/STEllAR-GROUP/hpx/pull/3530
[3428] https://github.com/STEllAR-GROUP/hpx/pull/3529
[3429] https://github.com/STEllAR-GROUP/hpx/pull/3528
[3430] https://github.com/STEllAR-GROUP/hpx/pull/3524
[3431] https://github.com/STEllAR-GROUP/hpx/pull/3523
[3432] https://github.com/STEllAR-GROUP/hpx/pull/3522
[3433] https://github.com/STEllAR-GROUP/hpx/pull/3521
[3434] https://github.com/STEllAR-GROUP/hpx/pull/3520
[3435] https://github.com/STEllAR-GROUP/hpx/pull/3519
[3436] https://github.com/STEllAR-GROUP/hpx/pull/3518

- PR #3516[3437] - Drop usage of deprecated facilities removed in C++17
- PR #3515[3438] - Remove remaining uses of Boost.TypeTraits
- PR #3513[3439] - Fixing a CMake problem when trying to use libfabric
- PR #3508[3440] - Remove memory_block component
- PR #3507[3441] - Propagating the MPI compile definitions to all relevant targets
- PR #3503[3442] - Update documentation colors and logo
- PR #3502[3443] - Fix bogus `throws` bindings in scheduled_thread_pool_impl
- PR #3501[3444] - Split parallel::remove_if tests to avoid OOM on CircleCI
- PR #3500[3445] - Support NONAMEPREFIX in add_hpx_library()
- PR #3497[3446] - Note that cuda support requires cmake 3.9
- PR #3495[3447] - Fixing dataflow
- PR #3493[3448] - Remove deprecated options for 1.2.0 part 2
- PR #3492[3449] - Add CUDA_LINK_LIBRARIES_KEYWORD to allow PRIVATE keyword in linkage t…
- PR #3491[3450] - Changing Base docker image
- PR #3490[3451] - Don't create tasks immediately with hpx::apply
- PR #3489[3452] - Remove deprecated options for 1.2.0
- PR #3488[3453] - Revert "Use BUILD_INTERFACE generator expression to fix cmake flag exports"
- PR #3487[3454] - Revert "Fixing type attribute warning for transfer_action"
- PR #3485[3455] - Use BUILD_INTERFACE generator expression to fix cmake flag exports
- PR #3483[3456] - Fixing type attribute warning for transfer_action
- PR #3481[3457] - Remove unused variables
- PR #3480[3458] - Towards a more lightweight transfer action
- PR #3479[3459] - Fix FLAGS - Use correct version of target_compile_options

3437 https://github.com/STEllAR-GROUP/hpx/pull/3516
3438 https://github.com/STEllAR-GROUP/hpx/pull/3515
3439 https://github.com/STEllAR-GROUP/hpx/pull/3513
3440 https://github.com/STEllAR-GROUP/hpx/pull/3508
3441 https://github.com/STEllAR-GROUP/hpx/pull/3507
3442 https://github.com/STEllAR-GROUP/hpx/pull/3503
3443 https://github.com/STEllAR-GROUP/hpx/pull/3502
3444 https://github.com/STEllAR-GROUP/hpx/pull/3501
3445 https://github.com/STEllAR-GROUP/hpx/pull/3500
3446 https://github.com/STEllAR-GROUP/hpx/pull/3497
3447 https://github.com/STEllAR-GROUP/hpx/pull/3495
3448 https://github.com/STEllAR-GROUP/hpx/pull/3493
3449 https://github.com/STEllAR-GROUP/hpx/pull/3492
3450 https://github.com/STEllAR-GROUP/hpx/pull/3491
3451 https://github.com/STEllAR-GROUP/hpx/pull/3490
3452 https://github.com/STEllAR-GROUP/hpx/pull/3489
3453 https://github.com/STEllAR-GROUP/hpx/pull/3488
3454 https://github.com/STEllAR-GROUP/hpx/pull/3487
3455 https://github.com/STEllAR-GROUP/hpx/pull/3485
3456 https://github.com/STEllAR-GROUP/hpx/pull/3483
3457 https://github.com/STEllAR-GROUP/hpx/pull/3481
3458 https://github.com/STEllAR-GROUP/hpx/pull/3480
3459 https://github.com/STEllAR-GROUP/hpx/pull/3479

- PR #3478[3460] - Making sure the application's exit code is properly propagated back to the OS
- PR #3476[3461] - Don't print docker credentials as part of the environment.
- PR #3473[3462] - Fixing invalid cmake code if no jemalloc prefix was given
- PR #3472[3463] - Attempting to work around recent clang test compilation failures
- PR #3471[3464] - Enable jemalloc on windows
- PR #3470[3465] - Updates readme
- PR #3468[3466] - Avoid hang if there is an exception thrown during startup
- PR #3467[3467] - Add compiler specific fallthrough attributes if C++17 attribute is not available
- PR #3466[3468] - - bugfix : fix compilation with llvm-7.0
- PR #3465[3469] - This patch adds various optimizations extracted from the thread_local_allocator work
- PR #3464[3470] - Check for forked repos in CircleCI docker push step
- PR #3463[3471] - - cmake : create the parent directory before symlinking
- PR #3459[3472] - Remove unused/incomplete functionality from util/logging
- PR #3458[3473] - Fix a problem with scope of CMAKE_CXX_FLAGS and hpx_add_compile_flag
- PR #3457[3474] - Fixing more size_t -> int16_t (and similar) warnings
- PR #3456[3475] - Add #ifdefs to topology.cpp to support old hwloc versions again
- PR #3454[3476] - Fixing warnings related to silent conversion of size_t –> int16_t
- PR #3451[3477] - Add examples as unit tests
- PR #3450[3478] - Constexpr-fying bind and other functional facilities
- PR #3446[3479] - Fix some thread suspension timeouts
- PR #3445[3480] - Fix various warnings
- PR #3443[3481] - Only enable service pool config options if pools are enabled
- PR #3441[3482] - Fix missing closing brackets in documentation

[3460] https://github.com/STEllAR-GROUP/hpx/pull/3478
[3461] https://github.com/STEllAR-GROUP/hpx/pull/3476
[3462] https://github.com/STEllAR-GROUP/hpx/pull/3473
[3463] https://github.com/STEllAR-GROUP/hpx/pull/3472
[3464] https://github.com/STEllAR-GROUP/hpx/pull/3471
[3465] https://github.com/STEllAR-GROUP/hpx/pull/3470
[3466] https://github.com/STEllAR-GROUP/hpx/pull/3468
[3467] https://github.com/STEllAR-GROUP/hpx/pull/3467
[3468] https://github.com/STEllAR-GROUP/hpx/pull/3466
[3469] https://github.com/STEllAR-GROUP/hpx/pull/3465
[3470] https://github.com/STEllAR-GROUP/hpx/pull/3464
[3471] https://github.com/STEllAR-GROUP/hpx/pull/3463
[3472] https://github.com/STEllAR-GROUP/hpx/pull/3459
[3473] https://github.com/STEllAR-GROUP/hpx/pull/3458
[3474] https://github.com/STEllAR-GROUP/hpx/pull/3457
[3475] https://github.com/STEllAR-GROUP/hpx/pull/3456
[3476] https://github.com/STEllAR-GROUP/hpx/pull/3454
[3477] https://github.com/STEllAR-GROUP/hpx/pull/3451
[3478] https://github.com/STEllAR-GROUP/hpx/pull/3450
[3479] https://github.com/STEllAR-GROUP/hpx/pull/3446
[3480] https://github.com/STEllAR-GROUP/hpx/pull/3445
[3481] https://github.com/STEllAR-GROUP/hpx/pull/3443
[3482] https://github.com/STEllAR-GROUP/hpx/pull/3441

- PR #3439[3483] - Use correct MPI CXX libraries for MPI parcelport
- PR #3436[3484] - Add projection function to find_* (and fix very bad bug)
- PR #3435[3485] - Fixing 1205
- PR #3434[3486] - Fix threads cores
- PR #3433[3487] - Add Heise Online to release announcement list
- PR #3432[3488] - Don't track task dependencies for distributed runs
- PR #3431[3489] - Circle CI setting changes for hpxMP
- PR #3430[3490] - Fix unused params warning
- PR #3429[3491] - One thread per core
- PR #3428[3492] - This suppresses a deprecation warning that is being issued by MSVC 19.15.26726
- PR #3427[3493] - Fixes #3426
- PR #3425[3494] - Use source cache and workspace between job steps on CircleCI
- PR #3421[3495] - Add CDash timing output to future overhead test (for graphs)
- PR #3420[3496] - Add guided_pool_executor
- PR #3419[3497] - Fix typo in CircleCI config
- PR #3418[3498] - Add sphinx documentation
- PR #3415[3499] - Scheduler NUMA hint and shared priority scheduler
- PR #3414[3500] - Adding step to synchronize the APEX release
- PR #3413[3501] - Fixing multiple defines of APEX_HAVE_HPX
- PR #3412[3502] - Fixes linking with libhpx_wrap error with BSD and Windows based systems
- PR #3410[3503] - Fix typo in CMakeLists.txt
- PR #3409[3504] - Fix brackets and indentation in existing_performance_counters.qbk
- PR #3407[3505] - Fix unused param and extra ; warnings emitted by gcc 8.x

3483 https://github.com/STEllAR-GROUP/hpx/pull/3439
3484 https://github.com/STEllAR-GROUP/hpx/pull/3436
3485 https://github.com/STEllAR-GROUP/hpx/pull/3435
3486 https://github.com/STEllAR-GROUP/hpx/pull/3434
3487 https://github.com/STEllAR-GROUP/hpx/pull/3433
3488 https://github.com/STEllAR-GROUP/hpx/pull/3432
3489 https://github.com/STEllAR-GROUP/hpx/pull/3431
3490 https://github.com/STEllAR-GROUP/hpx/pull/3430
3491 https://github.com/STEllAR-GROUP/hpx/pull/3429
3492 https://github.com/STEllAR-GROUP/hpx/pull/3428
3493 https://github.com/STEllAR-GROUP/hpx/pull/3427
3494 https://github.com/STEllAR-GROUP/hpx/pull/3425
3495 https://github.com/STEllAR-GROUP/hpx/pull/3421
3496 https://github.com/STEllAR-GROUP/hpx/pull/3420
3497 https://github.com/STEllAR-GROUP/hpx/pull/3419
3498 https://github.com/STEllAR-GROUP/hpx/pull/3418
3499 https://github.com/STEllAR-GROUP/hpx/pull/3415
3500 https://github.com/STEllAR-GROUP/hpx/pull/3414
3501 https://github.com/STEllAR-GROUP/hpx/pull/3413
3502 https://github.com/STEllAR-GROUP/hpx/pull/3412
3503 https://github.com/STEllAR-GROUP/hpx/pull/3410
3504 https://github.com/STEllAR-GROUP/hpx/pull/3409
3505 https://github.com/STEllAR-GROUP/hpx/pull/3407

- PR #3406[3506] - Adding thread local allocator and use it for future shared states
- PR #3405[3507] - Adding DHPX_HAVE_THREAD_LOCAL_STORAGE=ON to builds
- PR #3404[3508] - fixing multiple definition of main() in linux
- PR #3402[3509] - Allow debug option to be enabled only for Linux systems with dynamic main on
- PR #3401[3510] - Fix cuda_future_helper.h when compiling with C++11
- PR #3400[3511] - Fix floating point exception scheduler_base idle backoff
- PR #3398[3512] - Atomic future state
- PR #3397[3513] - Fixing code for older gcc versions
- PR #3396[3514] - Allowing to register thread event functions (start/stop/error)
- PR #3394[3515] - Fix small mistake in primary_namespace_server.cpp
- PR #3393[3516] - Explicitly instantiate configured schedulers
- PR #3392[3517] - Add performance counters background overhead and background work duration
- PR #3391[3518] - Adapt integration of HPXMP to latest build system changes
- PR #3390[3519] - Make AGAS measurements optional
- PR #3389[3520] - Fix deadlock during shutdown
- PR #3388[3521] - Add several functionalities allowing to optimize synchronous action invocation
- PR #3387[3522] - Add cmake option to opt out of fail-compile tests
- PR #3386[3523] - Adding support for boost::container::small_vector to dataflow
- PR #3385[3524] - Adds Debug option for hpx initializing from main
- PR #3384[3525] - This hopefully fixes two tests that occasionally fail
- PR #3383[3526] - Making sure thread local storage is enable for hpxMP
- PR #3382[3527] - Fix usage of HPX_CAPTURE together with default value capture [=]
- PR #3381[3528] - Replace undefined instantiations of uniform_int_distribution

[3506] https://github.com/STEllAR-GROUP/hpx/pull/3406
[3507] https://github.com/STEllAR-GROUP/hpx/pull/3405
[3508] https://github.com/STEllAR-GROUP/hpx/pull/3404
[3509] https://github.com/STEllAR-GROUP/hpx/pull/3402
[3510] https://github.com/STEllAR-GROUP/hpx/pull/3401
[3511] https://github.com/STEllAR-GROUP/hpx/pull/3400
[3512] https://github.com/STEllAR-GROUP/hpx/pull/3398
[3513] https://github.com/STEllAR-GROUP/hpx/pull/3397
[3514] https://github.com/STEllAR-GROUP/hpx/pull/3396
[3515] https://github.com/STEllAR-GROUP/hpx/pull/3394
[3516] https://github.com/STEllAR-GROUP/hpx/pull/3393
[3517] https://github.com/STEllAR-GROUP/hpx/pull/3392
[3518] https://github.com/STEllAR-GROUP/hpx/pull/3391
[3519] https://github.com/STEllAR-GROUP/hpx/pull/3390
[3520] https://github.com/STEllAR-GROUP/hpx/pull/3389
[3521] https://github.com/STEllAR-GROUP/hpx/pull/3388
[3522] https://github.com/STEllAR-GROUP/hpx/pull/3387
[3523] https://github.com/STEllAR-GROUP/hpx/pull/3386
[3524] https://github.com/STEllAR-GROUP/hpx/pull/3385
[3525] https://github.com/STEllAR-GROUP/hpx/pull/3384
[3526] https://github.com/STEllAR-GROUP/hpx/pull/3383
[3527] https://github.com/STEllAR-GROUP/hpx/pull/3382
[3528] https://github.com/STEllAR-GROUP/hpx/pull/3381

- PR #3380[3529] - Add missing semicolons to uses of HPX_COMPILER_FENCE
- PR #3379[3530] - Fixing #3378
- PR #3377[3531] - Adding build system support to integrate hpxmp into hpx at the user's machine
- PR #3375[3532] - Replacing wrapper for __libc_start_main with main
- PR #3374[3533] - Adds hpx_wrap to HPX_LINK_LIBRARIES which links only when specified.
- PR #3373[3534] - Forcing cache settings in HPXConfig.cmake to guarantee updated values
- PR #3372[3535] - Fix some more c++11 build problems
- PR #3371[3536] - Adds HPX_LINKER_FLAGS to HPX applications without editing their source codes
- PR #3370[3537] - util::format: add type_specifier<> specializations for %!s(MISSING) and %!l(MISSING)s
- PR #3369[3538] - Adding configuration option to allow explicit disable of the new hpx_main feature on Linux
- PR #3368[3539] - Updates doc with recent hpx_wrap implementation
- PR #3367[3540] - Adds Mac OS implementation to hpx_main.hpp
- PR #3365[3541] - Fix order of hpx libs in HPX_CONF_LIBRARIES.
- PR #3363[3542] - Apex fixing null wrapper
- PR #3361[3543] - Making sure all parcels get destroyed on an HPX thread (TCP pp)
- PR #3359[3544] - Feature/improveerrorforcompiler
- PR #3357[3545] - Static/dynamic executable implementation
- PR #3355[3546] - Reverting changes introduced by #3283 as those make applications hang
- PR #3354[3547] - Add external dependencies to HPX_LIBRARY_DIR
- PR #3353[3548] - Fix libfabric tcp
- PR #3351[3549] - Move obsolete header to tests directory.
- PR #3350[3550] - Renaming two functions to avoid problem described in #3285
- PR #3349[3551] - Make idle backoff exponential with maximum sleep time

[3529] https://github.com/STEllAR-GROUP/hpx/pull/3380
[3530] https://github.com/STEllAR-GROUP/hpx/pull/3379
[3531] https://github.com/STEllAR-GROUP/hpx/pull/3377
[3532] https://github.com/STEllAR-GROUP/hpx/pull/3375
[3533] https://github.com/STEllAR-GROUP/hpx/pull/3374
[3534] https://github.com/STEllAR-GROUP/hpx/pull/3373
[3535] https://github.com/STEllAR-GROUP/hpx/pull/3372
[3536] https://github.com/STEllAR-GROUP/hpx/pull/3371
[3537] https://github.com/STEllAR-GROUP/hpx/pull/3370
[3538] https://github.com/STEllAR-GROUP/hpx/pull/3369
[3539] https://github.com/STEllAR-GROUP/hpx/pull/3368
[3540] https://github.com/STEllAR-GROUP/hpx/pull/3367
[3541] https://github.com/STEllAR-GROUP/hpx/pull/3365
[3542] https://github.com/STEllAR-GROUP/hpx/pull/3363
[3543] https://github.com/STEllAR-GROUP/hpx/pull/3361
[3544] https://github.com/STEllAR-GROUP/hpx/pull/3359
[3545] https://github.com/STEllAR-GROUP/hpx/pull/3357
[3546] https://github.com/STEllAR-GROUP/hpx/pull/3355
[3547] https://github.com/STEllAR-GROUP/hpx/pull/3354
[3548] https://github.com/STEllAR-GROUP/hpx/pull/3353
[3549] https://github.com/STEllAR-GROUP/hpx/pull/3351
[3550] https://github.com/STEllAR-GROUP/hpx/pull/3350
[3551] https://github.com/STEllAR-GROUP/hpx/pull/3349

- PR #3347[3552] - Replace *simple_component** with *component** in the Documentation
- PR #3346[3553] - Fix CMakeLists.txt example in quick start
- PR #3345[3554] - Fix automatic setting of HPX_MORE_THAN_64_THREADS
- PR #3344[3555] - Reduce amount of information printed for unknown command line options
- PR #3343[3556] - Safeguard HPX against destruction in global contexts
- PR #3341[3557] - Allowing for all command line options to be used as configuration settings
- PR #3340[3558] - Always convert inspect results to JUnit XML
- PR #3336[3559] - Only run docker push on master on CircleCI
- PR #3335[3560] - Update description of hpx.os_threads config parameter.
- PR #3334[3561] - Making sure early logging settings don't get mixed with others
- PR #3333[3562] - Update CMake links and versions in documentation
- PR #3332[3563] - Add notes on target suffixes to CMake documentation
- PR #3331[3564] - Add quickstart section to documentation
- PR #3330[3565] - Rename resource_partitioner test to avoid conflicts with pseudodependencies
- PR #3328[3566] - Making sure object is pinned while executing actions, even if action returns a future
- PR #3327[3567] - Add missing std::forward to tuple.hpp
- PR #3326[3568] - Make sure logging is up and running while modules are being discovered.
- PR #3324[3569] - Replace C++14 overload of std::equal with C++11 code.
- PR #3323[3570] - Fix a missing apex thread data (wrapper) initialization
- PR #3320[3571] - Adding support for -std=c++2a (define *HPX_WITH_CXX2A=On*)
- PR #3319[3572] - Replacing C++14 feature with equivalent C++11 code
- PR #3317[3573] - Fix compilation with VS 15.7.1 and /std:c++latest
- PR #3316[3574] - Fix includes for 1d_stencil_*_omp examples

[3552] https://github.com/STEllAR-GROUP/hpx/pull/3347
[3553] https://github.com/STEllAR-GROUP/hpx/pull/3346
[3554] https://github.com/STEllAR-GROUP/hpx/pull/3345
[3555] https://github.com/STEllAR-GROUP/hpx/pull/3344
[3556] https://github.com/STEllAR-GROUP/hpx/pull/3343
[3557] https://github.com/STEllAR-GROUP/hpx/pull/3341
[3558] https://github.com/STEllAR-GROUP/hpx/pull/3340
[3559] https://github.com/STEllAR-GROUP/hpx/pull/3336
[3560] https://github.com/STEllAR-GROUP/hpx/pull/3335
[3561] https://github.com/STEllAR-GROUP/hpx/pull/3334
[3562] https://github.com/STEllAR-GROUP/hpx/pull/3333
[3563] https://github.com/STEllAR-GROUP/hpx/pull/3332
[3564] https://github.com/STEllAR-GROUP/hpx/pull/3331
[3565] https://github.com/STEllAR-GROUP/hpx/pull/3330
[3566] https://github.com/STEllAR-GROUP/hpx/pull/3328
[3567] https://github.com/STEllAR-GROUP/hpx/pull/3327
[3568] https://github.com/STEllAR-GROUP/hpx/pull/3326
[3569] https://github.com/STEllAR-GROUP/hpx/pull/3324
[3570] https://github.com/STEllAR-GROUP/hpx/pull/3323
[3571] https://github.com/STEllAR-GROUP/hpx/pull/3320
[3572] https://github.com/STEllAR-GROUP/hpx/pull/3319
[3573] https://github.com/STEllAR-GROUP/hpx/pull/3317
[3574] https://github.com/STEllAR-GROUP/hpx/pull/3316

- PR #3314[3575] - Remove some unused parameter warnings
- PR #3313[3576] - Fix pu-step and pu-offset command line options
- PR #3312[3577] - Add conversion of inspect reports to JUnit XML
- PR #3311[3578] - Fix escaping of closing braces in format specification syntax
- PR #3310[3579] - Don't overwrite user settings with defaults in registration database
- PR #3309[3580] - Fixing potential stack overflow for dataflow
- PR #3308[3581] - This updates the .clang-format configuration file to utilize newer features
- PR #3306[3582] - Marking migratable objects in their gid to allow not handling migration in AGAS
- PR #3305[3583] - Add proper exception handling to run_as_hpx_thread
- PR #3303[3584] - Changed std::rand to a better inbuilt PRNG Generator
- PR #3302[3585] - All non-migratable (simple) components now encode their lva and component type in their gid
- PR #3301[3586] - Add nullptr_t overloads to resource partitioner
- PR #3298[3587] - Apex task wrapper memory bug
- PR #3295[3588] - Fix mistakes after merge of CircleCI config
- PR #3294[3589] - Fix partitioned vector include in partitioned_vector_find tests
- PR #3293[3590] - Adding emplace support to promise and make_ready_future
- PR #3292[3591] - Add new cuda kernel synchronization with hpx::future demo
- PR #3291[3592] - Fixes #3290
- PR #3289[3593] - Fixing Docker image creation
- PR #3288[3594] - Avoid allocating shared state for wait_all
- PR #3287[3595] - Fixing /scheduler/utilization/instantaneous performance counter
- PR #3286[3596] - dataflow() and future::then() use sync policy where possible
- PR #3284[3597] - Background thread can use relaxed atomics to manipulate thread state

[3575] https://github.com/STEllAR-GROUP/hpx/pull/3314
[3576] https://github.com/STEllAR-GROUP/hpx/pull/3313
[3577] https://github.com/STEllAR-GROUP/hpx/pull/3312
[3578] https://github.com/STEllAR-GROUP/hpx/pull/3311
[3579] https://github.com/STEllAR-GROUP/hpx/pull/3310
[3580] https://github.com/STEllAR-GROUP/hpx/pull/3309
[3581] https://github.com/STEllAR-GROUP/hpx/pull/3308
[3582] https://github.com/STEllAR-GROUP/hpx/pull/3306
[3583] https://github.com/STEllAR-GROUP/hpx/pull/3305
[3584] https://github.com/STEllAR-GROUP/hpx/pull/3303
[3585] https://github.com/STEllAR-GROUP/hpx/pull/3302
[3586] https://github.com/STEllAR-GROUP/hpx/pull/3301
[3587] https://github.com/STEllAR-GROUP/hpx/pull/3298
[3588] https://github.com/STEllAR-GROUP/hpx/pull/3295
[3589] https://github.com/STEllAR-GROUP/hpx/pull/3294
[3590] https://github.com/STEllAR-GROUP/hpx/pull/3293
[3591] https://github.com/STEllAR-GROUP/hpx/pull/3292
[3592] https://github.com/STEllAR-GROUP/hpx/pull/3291
[3593] https://github.com/STEllAR-GROUP/hpx/pull/3289
[3594] https://github.com/STEllAR-GROUP/hpx/pull/3288
[3595] https://github.com/STEllAR-GROUP/hpx/pull/3287
[3596] https://github.com/STEllAR-GROUP/hpx/pull/3286
[3597] https://github.com/STEllAR-GROUP/hpx/pull/3284

- PR #3283[3598] - Do not unwrap ready future
- PR #3282[3599] - Fix virtual method override warnings in static schedulers
- PR #3281[3600] - Disable set_area_membind_nodeset for OSX
- PR #3279[3601] - Add two variations to the future_overhead benchmark
- PR #3278[3602] - Fix circleci workspace
- PR #3277[3603] - Support external plugins
- PR #3276[3604] - Fix missing parenthesis in hello_compute.cu.
- PR #3274[3605] - Reinit counters synchronously in reinit_counters test
- PR #3273[3606] - Splitting tests to avoid compiler OOM
- PR #3271[3607] - Remove leftover code from context_generic_context.hpp
- PR #3269[3608] - Fix bulk_construct with count = 0
- PR #3268[3609] - Replace constexpr with HPX_CXX14_CONSTEXPR and HPX_CONSTEXPR
- PR #3266[3610] - Replace boost::format with custom sprintf-based implementation
- PR #3265[3611] - Split parallel tests on CircleCI
- PR #3262[3612] - Making sure documentation correctly links to source files
- PR #3261[3613] - Apex refactoring fix rebind
- PR #3260[3614] - Isolate performance counter parser into a separate TU
- PR #3256[3615] - Post 1.1.0 version bumps
- PR #3254[3616] - Adding trait for actions allowing to make runtime decision on whether to execute it directly
- PR #3253[3617] - Bump minimal supported Boost to 1.58.0
- PR #3251[3618] - Adds new feature: changing interval used in interval_timer (issue 3244)
- PR #3239[3619] - Changing std::rand() to a better inbuilt PRNG generator.
- PR #3234[3620] - Disable background thread when networking is off

3598 https://github.com/STEllAR-GROUP/hpx/pull/3283
3599 https://github.com/STEllAR-GROUP/hpx/pull/3282
3600 https://github.com/STEllAR-GROUP/hpx/pull/3281
3601 https://github.com/STEllAR-GROUP/hpx/pull/3279
3602 https://github.com/STEllAR-GROUP/hpx/pull/3278
3603 https://github.com/STEllAR-GROUP/hpx/pull/3277
3604 https://github.com/STEllAR-GROUP/hpx/pull/3276
3605 https://github.com/STEllAR-GROUP/hpx/pull/3274
3606 https://github.com/STEllAR-GROUP/hpx/pull/3273
3607 https://github.com/STEllAR-GROUP/hpx/pull/3271
3608 https://github.com/STEllAR-GROUP/hpx/pull/3269
3609 https://github.com/STEllAR-GROUP/hpx/pull/3268
3610 https://github.com/STEllAR-GROUP/hpx/pull/3266
3611 https://github.com/STEllAR-GROUP/hpx/pull/3265
3612 https://github.com/STEllAR-GROUP/hpx/pull/3262
3613 https://github.com/STEllAR-GROUP/hpx/pull/3261
3614 https://github.com/STEllAR-GROUP/hpx/pull/3260
3615 https://github.com/STEllAR-GROUP/hpx/pull/3256
3616 https://github.com/STEllAR-GROUP/hpx/pull/3254
3617 https://github.com/STEllAR-GROUP/hpx/pull/3253
3618 https://github.com/STEllAR-GROUP/hpx/pull/3251
3619 https://github.com/STEllAR-GROUP/hpx/pull/3239
3620 https://github.com/STEllAR-GROUP/hpx/pull/3234

- PR #3232[3621] - Clean up suspension tests
- PR #3230[3622] - Add optional scheduler mode parameter to create_thread_pool function
- PR #3228[3623] - Allow suspension also on static schedulers
- PR #3163[3624] - libfabric parcelport w/o HPX_PARCELPORT_LIBFABRIC_ENDPOINT_RDM
- PR #3036[3625] - Switching to CircleCI 2.0

HPX V1.1.0 (Mar 24, 2018)

General changes

Here are some of the main highlights and changes for this release (in no particular order):

- We have changed the way *HPX* manages the processing units on a node. We do not longer implicitly bind all available cores to a single thread pool. The user has now full control over what processing units are bound to what thread pool, each with a separate scheduler. It is now also possible to create your own scheduler implementation and control what processing units this scheduler should use. We added the `hpx::resource::partitioner` that manages all available processing units and assigns resources to the used thread pools. Thread pools can be now be suspended/resumed independently. This functionality helps in running *HPX* concurrently to code that is directly relying on OpenMP[3626] and/or MPI[3627].
- We have continued to implement various parallel algorithms. *HPX* now almost completely implements all of the parallel algorithms as specified by the C++17 Standard[3628]. We have also continued to implement these algorithms for the distributed use case (for segmented data structures, such as `hpx::partitioned_vector`).
- Added a compatibility layer for `std::thread`, `std::mutex`, and `std::condition_variable` allowing for the code to use those facilities where available and to fall back to the corresponding Boost facilities otherwise. The CMake[3629] configuration option `-DHPX_WITH_THREAD_COMPATIBILITY=On` can be used to force using the Boost equivalents.
- The parameter sequence for the `hpx::parallel::transform_inclusive_scan` overload taking one iterator range has changed (again) to match the changes this algorithm has undergone while being moved to C++17. The old overloads can be still enabled at configure time by passing `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake[3630].
- The parameter sequence for the `hpx::parallel::inclusive_scan` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17. The old overloads can be still enabled at configure time by passing `-DHPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY=On` to CMake.
- Added a helper facility `hpx::local_new` which is equivalent to `hpx::new_` except that it creates components locally only. As a consequence, the used component constructor may accept non-serializable argument types and/or non-const references or pointers.
- Removed the (broken) component type `hpx::lcos::queue<T>`. The old type is still available at configure time by passing `-DHPX_WITH_QUEUE_COMPATIBILITY=On` to CMake.

[3621] https://github.com/STEllAR-GROUP/hpx/pull/3232
[3622] https://github.com/STEllAR-GROUP/hpx/pull/3230
[3623] https://github.com/STEllAR-GROUP/hpx/pull/3228
[3624] https://github.com/STEllAR-GROUP/hpx/pull/3163
[3625] https://github.com/STEllAR-GROUP/hpx/pull/3036
[3626] https://openmp.org/wp/
[3627] https://en.wikipedia.org/wiki/Message_Passing_Interface
[3628] http://www.open-std.org/jtc1/sc22/wg21
[3629] https://www.cmake.org
[3630] https://www.cmake.org

- The parallel algorithms adopted for C++17 restrict the iterator categories usable with those to at least forward iterators. Our implementation of the parallel algorithms was supporting input iterators (and output iterators) as well by simply falling back to sequential execution. We have now made our implementations conforming by requiring at least forward iterators. In order to enable the old behavior use the compatibility option `-DHPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT=On` on the CMake[3631] command line.
- We have added the functionalities allowing for LCOs being implemented using (simple) components. Before LCOs had to always be implemented using managed components.
- User defined components don't have to be default-constructible anymore. Return types from actions don't have to be default-constructible anymore either. Our serialization layer now in general supports non-default-constructible types.
- We have added a new launch policy `hpx::launch::lazy` that allows oneto defer the decision on what launch policy to use to the point of execution. This policy is initialized with a function (object) that – when invoked – is expected to produce the desired launch policy.

Breaking changes

- We have dropped support for the gcc compiler version V4.8. The minimal gcc version we now test on is gcc V4.9. The minimally required version of CMake[3632] is now V3.3.2.
- We have dropped support for the Visual Studio 2013 compiler version. The minimal Visual Studio version we now test on is Visual Studio 2015.5.
- We have dropped support for the Boost V1.51-V1.54. The minimal version of Boost we now test is Boost V1.55.
- We have dropped support for the `hpx::util::unwrapped` API. `hpx::util::unwrapped` will stay functional to some degree, until it finally gets removed in a later version of HPX. The functional usage of `hpx::util::unwrapped` should be changed to the new `hpx::util::unwrapping` function whereas the immediate usage should be replaced to `hpx::util::unwrap`.
- The performance counter names referring to properties as exposed by the threading subsystem have changes as those now additionally have to specify the thread-pool. See the corresponding documentation for more details.
- The overloads of `hpx::async` that invoke an action do not perform implicit unwrapping of the returned future anymore in case the invoked function does return a future in the first place. In this case `hpx::async` now returns a `hpx::future<future<T>>` making its behavior conforming to its local counterpart.
- We have replaced the use of `boost::exception_ptr` in our APIs with the equivalent `std::exception_ptr`. Please change your codes accordingly. No compatibility settings are provided.
- We have removed the compatibility settings for `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` and `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY` as their life-cycle has reached its end.
- We have removed the experimental thread schedulers hierarchy_scheduler, periodic_priority_scheduler and throttling_scheduler in an effort to clean up and consolidate our thread schedulers.

[3631] https://www.cmake.org
[3632] https://www.cmake.org

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #3250[3633] - Apex refactoring with guids
- PR #3249[3634] - Updating People.qbk
- PR #3246[3635] - Assorted fixes for CUDA
- PR #3245[3636] - Apex refactoring with guids
- PR #3242[3637] - Modify task counting in thread_queue.hpp
- PR #3240[3638] - Fixed typos
- PR #3238[3639] - Readding accidentally removed std::abort
- PR #3237[3640] - Adding Pipeline example
- PR #3236[3641] - Fixing memory_block
- PR #3233[3642] - Make schedule_thread take suspended threads into account
- Issue #3226[3643] - memory_block is breaking, signaling SIGSEGV on a thread on creation and freeing
- PR #3225[3644] - Applying quick fix for hwloc-2.0
- Issue #3224[3645] - HPX counters crashing the application
- PR #3223[3646] - Fix returns when setting config entries
- Issue #3222[3647] - Errors linking libhpx.so
- Issue #3221[3648] - HPX on Mac OS X with HWLoc 2.0.0 fails to run
- PR #3216[3649] - Reorder a variadic array to satisfy VS 2017 15.6
- PR #3214[3650] - Changed prerequisites.qbk to avoid confusion while building boost
- PR #3213[3651] - Relax locks for thread suspension to avoid holding locks when yielding
- PR #3212[3652] - Fix check in sequenced_executor test
- PR #3211[3653] - Use preinit_array to set argc/argv in init_globally example

[3633] https://github.com/STEllAR-GROUP/hpx/pull/3250
[3634] https://github.com/STEllAR-GROUP/hpx/pull/3249
[3635] https://github.com/STEllAR-GROUP/hpx/pull/3246
[3636] https://github.com/STEllAR-GROUP/hpx/pull/3245
[3637] https://github.com/STEllAR-GROUP/hpx/pull/3242
[3638] https://github.com/STEllAR-GROUP/hpx/pull/3240
[3639] https://github.com/STEllAR-GROUP/hpx/pull/3238
[3640] https://github.com/STEllAR-GROUP/hpx/pull/3237
[3641] https://github.com/STEllAR-GROUP/hpx/pull/3236
[3642] https://github.com/STEllAR-GROUP/hpx/pull/3233
[3643] https://github.com/STEllAR-GROUP/hpx/issues/3226
[3644] https://github.com/STEllAR-GROUP/hpx/pull/3225
[3645] https://github.com/STEllAR-GROUP/hpx/issues/3224
[3646] https://github.com/STEllAR-GROUP/hpx/pull/3223
[3647] https://github.com/STEllAR-GROUP/hpx/issues/3222
[3648] https://github.com/STEllAR-GROUP/hpx/issues/3221
[3649] https://github.com/STEllAR-GROUP/hpx/pull/3216
[3650] https://github.com/STEllAR-GROUP/hpx/pull/3214
[3651] https://github.com/STEllAR-GROUP/hpx/pull/3213
[3652] https://github.com/STEllAR-GROUP/hpx/pull/3212
[3653] https://github.com/STEllAR-GROUP/hpx/pull/3211

- PR #3210[3654] - Adapted parallel::{search | search_n} for Ranges TS (see #1668)
- PR #3209[3655] - Fix locking problems during shutdown
- Issue #3208[3656] - init_globally throwing a run-time error
- PR #3206[3657] - Addition of new arithmetic performance counter "Count"
- PR #3205[3658] - Fixing return type calculation for bulk_then_execute
- PR #3204[3659] - Changing std::rand() to a better inbuilt PRNG generator
- PR #3203[3660] - Resolving problems during shutdown for VS2015
- PR #3202[3661] - Making sure resource partitioner is not accessed if its not valid
- PR #3201[3662] - Fixing optional::swap
- Issue #3200[3663] - hpx::util::optional fails
- PR #3199[3664] - Fix sliding_semaphore test
- PR #3198[3665] - Set pre_main status before launching run_helper
- PR #3197[3666] - Update README.rst
- PR #3194[3667] - parallel::{fill|fill_n} updated for Ranges TS
- PR #3193[3668] - Updating Runtime.cpp by adding correct description of Performance counters during register
- PR #3191[3669] - Fix sliding_semaphore_2338 test
- PR #3190[3670] - Topology improvements
- PR #3189[3671] - Deleting one include of median from BOOST library to arithmetics_counter file
- PR #3188[3672] - Optionally disable printing of diagnostics during terminate
- PR #3187[3673] - Suppressing cmake warning issued by cmake > V3.11
- PR #3185[3674] - Remove unused scoped_unlock, unlock_guard_try
- PR #3184[3675] - Fix nqueen example
- PR #3183[3676] - Add runtime start/stop, resume/suspend and OpenMP benchmarks

[3654] https://github.com/STEllAR-GROUP/hpx/pull/3210
[3655] https://github.com/STEllAR-GROUP/hpx/pull/3209
[3656] https://github.com/STEllAR-GROUP/hpx/issues/3208
[3657] https://github.com/STEllAR-GROUP/hpx/pull/3206
[3658] https://github.com/STEllAR-GROUP/hpx/pull/3205
[3659] https://github.com/STEllAR-GROUP/hpx/pull/3204
[3660] https://github.com/STEllAR-GROUP/hpx/pull/3203
[3661] https://github.com/STEllAR-GROUP/hpx/pull/3202
[3662] https://github.com/STEllAR-GROUP/hpx/pull/3201
[3663] https://github.com/STEllAR-GROUP/hpx/issues/3200
[3664] https://github.com/STEllAR-GROUP/hpx/pull/3199
[3665] https://github.com/STEllAR-GROUP/hpx/pull/3198
[3666] https://github.com/STEllAR-GROUP/hpx/pull/3197
[3667] https://github.com/STEllAR-GROUP/hpx/pull/3194
[3668] https://github.com/STEllAR-GROUP/hpx/pull/3193
[3669] https://github.com/STEllAR-GROUP/hpx/pull/3191
[3670] https://github.com/STEllAR-GROUP/hpx/pull/3190
[3671] https://github.com/STEllAR-GROUP/hpx/pull/3189
[3672] https://github.com/STEllAR-GROUP/hpx/pull/3188
[3673] https://github.com/STEllAR-GROUP/hpx/pull/3187
[3674] https://github.com/STEllAR-GROUP/hpx/pull/3185
[3675] https://github.com/STEllAR-GROUP/hpx/pull/3184
[3676] https://github.com/STEllAR-GROUP/hpx/pull/3183

- Issue #3182[3677] - bulk_then_execute has unexpected return type/does not compile
- Issue #3181[3678] - hwloc 2.0 breaks topo class and cannot be used
- Issue #3180[3679] - Schedulers that don't support suspend/resume are unusable
- PR #3179[3680] - Various minor changes to support FLeCSI
- PR #3178[3681] - Fix #3124
- PR #3177[3682] - Removed allgather
- PR #3176[3683] - Fixed Documentation for "using_hpx_pkgconfig"
- PR #3174[3684] - Add hpx::iostreams::ostream overload to format_to
- PR #3172[3685] - Fix lifo queue backend
- PR #3171[3686] - adding the missing unset() function to cpu_mask() for case of more than 64 threads
- PR #3170[3687] - Add cmake flag -DHPX_WITH_FAULT_TOLERANCE=ON (OFF by default)
- PR #3169[3688] - Adapted parallel::{count|count_if} for Ranges TS (see #1668)
- PR #3168[3689] - Changing used namespace for seq execution policy
- Issue #3167[3690] - Update GSoC projects
- Issue #3166[3691] - Application (Octotiger) gets stuck on hpx::finalize when only using one thread
- Issue #3165[3692] - Compilation of parallel algorithms with HPX_WITH_DATAPAR is broken
- PR #3164[3693] - Fixing component migration
- PR #3162[3694] - regex_from_pattern: escape regex special characters to avoid misinterpretation
- Issue #3161[3695] - Building HPX with hwloc 2.0.0 fails
- PR #3160[3696] - Fixing the handling of quoted command line arguments.
- PR #3158[3697] - Fixing a race with timed suspension (second attempt)
- PR #3157[3698] - Revert "Fixing a race with timed suspension"
- PR #3156[3699] - Fixing serialization of classes with incompatible serialize signature

[3677] https://github.com/STEllAR-GROUP/hpx/issues/3182
[3678] https://github.com/STEllAR-GROUP/hpx/issues/3181
[3679] https://github.com/STEllAR-GROUP/hpx/issues/3180
[3680] https://github.com/STEllAR-GROUP/hpx/pull/3179
[3681] https://github.com/STEllAR-GROUP/hpx/pull/3178
[3682] https://github.com/STEllAR-GROUP/hpx/pull/3177
[3683] https://github.com/STEllAR-GROUP/hpx/pull/3176
[3684] https://github.com/STEllAR-GROUP/hpx/pull/3174
[3685] https://github.com/STEllAR-GROUP/hpx/pull/3172
[3686] https://github.com/STEllAR-GROUP/hpx/pull/3171
[3687] https://github.com/STEllAR-GROUP/hpx/pull/3170
[3688] https://github.com/STEllAR-GROUP/hpx/pull/3169
[3689] https://github.com/STEllAR-GROUP/hpx/pull/3168
[3690] https://github.com/STEllAR-GROUP/hpx/issues/3167
[3691] https://github.com/STEllAR-GROUP/hpx/issues/3166
[3692] https://github.com/STEllAR-GROUP/hpx/issues/3165
[3693] https://github.com/STEllAR-GROUP/hpx/pull/3164
[3694] https://github.com/STEllAR-GROUP/hpx/pull/3162
[3695] https://github.com/STEllAR-GROUP/hpx/issues/3161
[3696] https://github.com/STEllAR-GROUP/hpx/pull/3160
[3697] https://github.com/STEllAR-GROUP/hpx/pull/3158
[3698] https://github.com/STEllAR-GROUP/hpx/pull/3157
[3699] https://github.com/STEllAR-GROUP/hpx/pull/3156

- PR #3154[3700] - More refactorings based on clang-tidy reports
- PR #3153[3701] - Fixing a race with timed suspension
- PR #3152[3702] - Documentation for runtime suspension
- PR #3151[3703] - Use small_vector only from boost version 1.59 onwards
- PR #3150[3704] - Avoiding more stack overflows
- PR #3148[3705] - Refactoring component_base and base_action/transfer_base_action
- PR #3147[3706] - Move yield_while out of detail namespace and into own file
- PR #3145[3707] - Remove a leftover of the cxx11 std array cleanup
- PR #3144[3708] - Minor changes to how actions are executed
- PR #3143[3709] - Fix stack overhead
- PR #3142[3710] - Fix typo in config.hpp
- PR #3141[3711] - Fixing small_vector compatibility with older boost version
- PR #3140[3712] - is_heap_text fix
- Issue #3139[3713] - Error in is_heap_tests.hpp
- PR #3138[3714] - Partially reverting #3126
- PR #3137[3715] - Suspend speedup
- PR #3136[3716] - Revert "Fixing #2325"
- PR #3135[3717] - Improving destruction of threads
- Issue #3134[3718] - HPX_SERIALIZATION_SPLIT_FREE does not stop compiler from looking for serialize() method
- PR #3133[3719] - Make hwloc compulsory
- PR #3132[3720] - Update CXX14 constexpr feature test
- PR #3131[3721] - Fixing #2325
- PR #3130[3722] - Avoid completion handler allocation

[3700] https://github.com/STEllAR-GROUP/hpx/pull/3154
[3701] https://github.com/STEllAR-GROUP/hpx/pull/3153
[3702] https://github.com/STEllAR-GROUP/hpx/pull/3152
[3703] https://github.com/STEllAR-GROUP/hpx/pull/3151
[3704] https://github.com/STEllAR-GROUP/hpx/pull/3150
[3705] https://github.com/STEllAR-GROUP/hpx/pull/3148
[3706] https://github.com/STEllAR-GROUP/hpx/pull/3147
[3707] https://github.com/STEllAR-GROUP/hpx/pull/3145
[3708] https://github.com/STEllAR-GROUP/hpx/pull/3144
[3709] https://github.com/STEllAR-GROUP/hpx/pull/3143
[3710] https://github.com/STEllAR-GROUP/hpx/pull/3142
[3711] https://github.com/STEllAR-GROUP/hpx/pull/3141
[3712] https://github.com/STEllAR-GROUP/hpx/pull/3140
[3713] https://github.com/STEllAR-GROUP/hpx/issues/3139
[3714] https://github.com/STEllAR-GROUP/hpx/pull/3138
[3715] https://github.com/STEllAR-GROUP/hpx/pull/3137
[3716] https://github.com/STEllAR-GROUP/hpx/pull/3136
[3717] https://github.com/STEllAR-GROUP/hpx/pull/3135
[3718] https://github.com/STEllAR-GROUP/hpx/issues/3134
[3719] https://github.com/STEllAR-GROUP/hpx/pull/3133
[3720] https://github.com/STEllAR-GROUP/hpx/pull/3132
[3721] https://github.com/STEllAR-GROUP/hpx/pull/3131
[3722] https://github.com/STEllAR-GROUP/hpx/pull/3130

- PR #3129[3723] - Suspend runtime
- PR #3128[3724] - Make docbook dtd and xsl path names consistent
- PR #3127[3725] - Add hpx::start nullptr overloads
- PR #3126[3726] - Cleaning up coroutine implementation
- PR #3125[3727] - Replacing nullptr with hpx::threads::invalid_thread_id
- Issue #3124[3728] - Add hello_world_component to CI builds
- PR #3123[3729] - Add new constructor.
- PR #3122[3730] - Fixing #3121
- Issue #3121[3731] - HPX_SMT_PAUSE is broken on non-x86 platforms when __GNUC__ is defined
- PR #3120[3732] - Don't use boost::intrusive_ptr for thread_id_type
- PR #3119[3733] - Disable default executor compatibility with V1 executors
- PR #3118[3734] - Adding performance_counter::reinit to allow for dynamically changing counter sets
- PR #3117[3735] - Replace uses of boost/experimental::optional with util::optional
- PR #3116[3736] - Moving background thread APEX timer #2980
- PR #3115[3737] - Fixing race condition in channel test
- PR #3114[3738] - Avoid using util::function for thread function wrappers
- PR #3113[3739] - cmake V3.10.2 has changed the variable names used for MPI
- PR #3112[3740] - Minor fixes to exclusive_scan algorithm
- PR #3111[3741] - Revert "fix detection of cxx11_std_atomic"
- PR #3110[3742] - Suspend thread pool
- PR #3109[3743] - Fixing thread scheduling when yielding a thread id
- PR #3108[3744] - Revert "Suspend thread pool"
- PR #3107[3745] - Remove UB from thread::id relational operators

[3723] https://github.com/STEllAR-GROUP/hpx/pull/3129
[3724] https://github.com/STEllAR-GROUP/hpx/pull/3128
[3725] https://github.com/STEllAR-GROUP/hpx/pull/3127
[3726] https://github.com/STEllAR-GROUP/hpx/pull/3126
[3727] https://github.com/STEllAR-GROUP/hpx/pull/3125
[3728] https://github.com/STEllAR-GROUP/hpx/issues/3124
[3729] https://github.com/STEllAR-GROUP/hpx/pull/3123
[3730] https://github.com/STEllAR-GROUP/hpx/pull/3122
[3731] https://github.com/STEllAR-GROUP/hpx/issues/3121
[3732] https://github.com/STEllAR-GROUP/hpx/pull/3120
[3733] https://github.com/STEllAR-GROUP/hpx/pull/3119
[3734] https://github.com/STEllAR-GROUP/hpx/pull/3118
[3735] https://github.com/STEllAR-GROUP/hpx/pull/3117
[3736] https://github.com/STEllAR-GROUP/hpx/pull/3116
[3737] https://github.com/STEllAR-GROUP/hpx/pull/3115
[3738] https://github.com/STEllAR-GROUP/hpx/pull/3114
[3739] https://github.com/STEllAR-GROUP/hpx/pull/3113
[3740] https://github.com/STEllAR-GROUP/hpx/pull/3112
[3741] https://github.com/STEllAR-GROUP/hpx/pull/3111
[3742] https://github.com/STEllAR-GROUP/hpx/pull/3110
[3743] https://github.com/STEllAR-GROUP/hpx/pull/3109
[3744] https://github.com/STEllAR-GROUP/hpx/pull/3108
[3745] https://github.com/STEllAR-GROUP/hpx/pull/3107

- PR #3106[3746] - Add cmake test for std::decay_t to fix cuda build
- PR #3105[3747] - Fixing refcount for async traversal frame
- PR #3104[3748] - Local execution of direct actions is now actually performed directly
- PR #3103[3749] - Adding support for generic counter_raw_values performance counter type
- Issue #3102[3750] - Introduce generic performance counter type returning an array of values
- PR #3101[3751] - Revert "Adapting stack overhead limit for gcc 4.9"
- PR #3100[3752] - Fix #3068 (condition_variable deadlock)
- PR #3099[3753] - Fixing lock held during suspension in papi counter component
- PR #3098[3754] - Unbreak broadcast_wait_for_2822 test
- PR #3097[3755] - Adapting stack overhead limit for gcc 4.9
- PR #3096[3756] - fix detection of cxx11_std_atomic
- PR #3095[3757] - Add ciso646 header to get _LIBCPP_VERSION for testing inplace merge
- PR #3094[3758] - Relax atomic operations on performance counter values
- PR #3093[3759] - Short-circuit all_of/any_of/none_of instantiations
- PR #3092[3760] - Take advantage of C++14 lambda capture initialization syntax, where possible
- PR #3091[3761] - Remove more references to Boost from logging code
- PR #3090[3762] - Unify use of yield/yield_k
- PR #3089[3763] - Fix a strange thing in parallel::detail::handle_exception. (Fix #2834.)
- Issue #3088[3764] - A strange thing in parallel::sort.
- PR #3087[3765] - Fixing assertion in default_distribution_policy
- PR #3086[3766] - Implement parallel::remove and parallel::remove_if
- PR #3085[3767] - Addressing breaking changes in Boost V1.66
- PR #3084[3768] - Ignore build warnings round 2

[3746] https://github.com/STEllAR-GROUP/hpx/pull/3106
[3747] https://github.com/STEllAR-GROUP/hpx/pull/3105
[3748] https://github.com/STEllAR-GROUP/hpx/pull/3104
[3749] https://github.com/STEllAR-GROUP/hpx/pull/3103
[3750] https://github.com/STEllAR-GROUP/hpx/issues/3102
[3751] https://github.com/STEllAR-GROUP/hpx/pull/3101
[3752] https://github.com/STEllAR-GROUP/hpx/pull/3100
[3753] https://github.com/STEllAR-GROUP/hpx/pull/3099
[3754] https://github.com/STEllAR-GROUP/hpx/pull/3098
[3755] https://github.com/STEllAR-GROUP/hpx/pull/3097
[3756] https://github.com/STEllAR-GROUP/hpx/pull/3096
[3757] https://github.com/STEllAR-GROUP/hpx/pull/3095
[3758] https://github.com/STEllAR-GROUP/hpx/pull/3094
[3759] https://github.com/STEllAR-GROUP/hpx/pull/3093
[3760] https://github.com/STEllAR-GROUP/hpx/pull/3092
[3761] https://github.com/STEllAR-GROUP/hpx/pull/3091
[3762] https://github.com/STEllAR-GROUP/hpx/pull/3090
[3763] https://github.com/STEllAR-GROUP/hpx/pull/3089
[3764] https://github.com/STEllAR-GROUP/hpx/issues/3088
[3765] https://github.com/STEllAR-GROUP/hpx/pull/3087
[3766] https://github.com/STEllAR-GROUP/hpx/pull/3086
[3767] https://github.com/STEllAR-GROUP/hpx/pull/3085
[3768] https://github.com/STEllAR-GROUP/hpx/pull/3084

- PR #3083[3769] - Fix typo HPX_WITH_MM_PREFECTH
- PR #3081[3770] - Pre-decay template arguments early
- PR #3080[3771] - Suspend thread pool
- PR #3079[3772] - Ignore build warnings
- PR #3078[3773] - Don't test inplace_merge with libc++
- PR #3076[3774] - Fixing 3075: Part 1
- PR #3074[3775] - Fix more build warnings
- PR #3073[3776] - Suspend thread cleanup
- PR #3072[3777] - Change existing symbol_namespace::iterate to return all data instead of invoking a callback
- PR #3071[3778] - Fixing pack_traversal_async test
- PR #3070[3779] - Fix dynamic_counters_loaded_1508 test by adding dependency to memory_component
- PR #3069[3780] - Fix scheduling loop exit
- Issue #3068[3781] - hpx::lcos::condition_variable could be suspect to deadlocks
- PR #3067[3782] - #ifdef out random_shuffle deprecated in later c++
- PR #3066[3783] - Make coalescing test depend on coalescing library to ensure it gets built
- PR #3065[3784] - Workaround for minimal_timed_async_executor_test compilation failures, attempts to copy a deferred call (in unevaluated context)
- PR #3064[3785] - Fixing wrong condition in wrapper_heap
- PR #3062[3786] - Fix exception handling for execution::seq
- PR #3061[3787] - Adapt MSVC C++ mode handling to VS15.5
- PR #3060[3788] - Fix compiler problem in MSVC release mode
- PR #3059[3789] - Fixing #2931
- Issue #3058[3790] - minimal_timed_async_executor_test_exe fails to compile on master (d6f505c)
- PR #3057[3791] - Fix stable_merge_2964 compilation problems

3769 https://github.com/STEllAR-GROUP/hpx/pull/3083
3770 https://github.com/STEllAR-GROUP/hpx/pull/3081
3771 https://github.com/STEllAR-GROUP/hpx/pull/3080
3772 https://github.com/STEllAR-GROUP/hpx/pull/3079
3773 https://github.com/STEllAR-GROUP/hpx/pull/3078
3774 https://github.com/STEllAR-GROUP/hpx/pull/3076
3775 https://github.com/STEllAR-GROUP/hpx/pull/3074
3776 https://github.com/STEllAR-GROUP/hpx/pull/3073
3777 https://github.com/STEllAR-GROUP/hpx/pull/3072
3778 https://github.com/STEllAR-GROUP/hpx/pull/3071
3779 https://github.com/STEllAR-GROUP/hpx/pull/3070
3780 https://github.com/STEllAR-GROUP/hpx/pull/3069
3781 https://github.com/STEllAR-GROUP/hpx/issues/3068
3782 https://github.com/STEllAR-GROUP/hpx/pull/3067
3783 https://github.com/STEllAR-GROUP/hpx/pull/3066
3784 https://github.com/STEllAR-GROUP/hpx/pull/3065
3785 https://github.com/STEllAR-GROUP/hpx/pull/3064
3786 https://github.com/STEllAR-GROUP/hpx/pull/3062
3787 https://github.com/STEllAR-GROUP/hpx/pull/3061
3788 https://github.com/STEllAR-GROUP/hpx/pull/3060
3789 https://github.com/STEllAR-GROUP/hpx/pull/3059
3790 https://github.com/STEllAR-GROUP/hpx/issues/3058
3791 https://github.com/STEllAR-GROUP/hpx/pull/3057

- PR #3056[3792] - Fix some build warnings caused by unused variables/unnecessary tests
- PR #3055[3793] - Update documentation for running tests
- Issue #3054[3794] - Assertion failure when using bulk hpx::**new_** in asynchronous mode
- PR #3052[3795] - Do not bind test running to cmake test build rule
- PR #3051[3796] - Fix HPX-Qt interaction in Qt example.
- Issue #3048[3797] - nqueen example fails occasionally
- PR #3047[3798] - Fixing #3044
- PR #3046[3799] - Add OS thread suspension
- PR #3042[3800] - PyCicle - first attempt at a build toold for checking PR's
- PR #3041[3801] - Fix a problem about asynchronous execution of parallel::merge and parallel::partition.
- PR #3040[3802] - Fix a mistake about exception handling in asynchronous execution of scan_partitioner.
- PR #3039[3803] - Consistently use executors to schedule work
- PR #3038[3804] - Fixing local direct function execution and lambda actions perfect forwarding
- PR #3035[3805] - Make parallel unit test names match build target/folder names
- PR #3033[3806] - Fix setting of default build type
- Issue #3032[3807] - Fix partitioner arg copy found in #2982
- Issue #3031[3808] - Errors linking libhpx.so due to missing references (master branch, commit 6679a8882)
- PR #3030[3809] - Revert "implement executor then interface with && forwarding reference"
- PR #3029[3810] - Run CI inspect checks before building
- PR #3028[3811] - Added range version of parallel::move
- Issue #3027[3812] - Implement all scheduling APIs in terms of executors
- PR #3026[3813] - implement executor then interface with && forwarding reference
- PR #3025[3814] - Fix typo unitialized to uninitialized

3792 https://github.com/STEllAR-GROUP/hpx/pull/3056
3793 https://github.com/STEllAR-GROUP/hpx/pull/3055
3794 https://github.com/STEllAR-GROUP/hpx/issues/3054
3795 https://github.com/STEllAR-GROUP/hpx/pull/3052
3796 https://github.com/STEllAR-GROUP/hpx/pull/3051
3797 https://github.com/STEllAR-GROUP/hpx/issues/3048
3798 https://github.com/STEllAR-GROUP/hpx/pull/3047
3799 https://github.com/STEllAR-GROUP/hpx/pull/3046
3800 https://github.com/STEllAR-GROUP/hpx/pull/3042
3801 https://github.com/STEllAR-GROUP/hpx/pull/3041
3802 https://github.com/STEllAR-GROUP/hpx/pull/3040
3803 https://github.com/STEllAR-GROUP/hpx/pull/3039
3804 https://github.com/STEllAR-GROUP/hpx/pull/3038
3805 https://github.com/STEllAR-GROUP/hpx/pull/3035
3806 https://github.com/STEllAR-GROUP/hpx/pull/3033
3807 https://github.com/STEllAR-GROUP/hpx/issues/3032
3808 https://github.com/STEllAR-GROUP/hpx/issues/3031
3809 https://github.com/STEllAR-GROUP/hpx/pull/3030
3810 https://github.com/STEllAR-GROUP/hpx/pull/3029
3811 https://github.com/STEllAR-GROUP/hpx/pull/3028
3812 https://github.com/STEllAR-GROUP/hpx/issues/3027
3813 https://github.com/STEllAR-GROUP/hpx/pull/3026
3814 https://github.com/STEllAR-GROUP/hpx/pull/3025

- PR #3024[3815] - Inspect fixes
- PR #3023[3816] - P0356 Simplified partial function application
- PR #3022[3817] - Master fixes
- PR #3021[3818] - Segfault fix
- PR #3020[3819] - Disable command-line aliasing for applications that use user_main
- PR #3019[3820] - Adding enable_elasticity option to pool configuration
- PR #3018[3821] - Fix stack overflow detection configuration in header files
- PR #3017[3822] - Speed up local action execution
- PR #3016[3823] - Unify stack-overflow detection options, remove reference to libsigsegv
- PR #3015[3824] - Speeding up accessing the resource partitioner and the topology info
- Issue #3014[3825] - HPX does not compile on POWER8 with gcc 5.4
- Issue #3013[3826] - hello_world occasionally prints multiple lines from a single OS-thread
- PR #3012[3827] - Silence warning about casting away qualifiers in itt_notify.hpp
- PR #3011[3828] - Fix cpuset leak in hwloc_topology_info.cpp
- PR #3010[3829] - Remove useless decay_copy
- PR #3009[3830] - Fixing 2996
- PR #3008[3831] - Remove unused internal function
- PR #3007[3832] - Fixing wrapper_heap alignment problems
- Issue #3006[3833] - hwloc memory leak
- PR #3004[3834] - Silence C4251 (needs to have dll-interface) for future_data_void
- Issue #3003[3835] - Suspension of runtime
- PR #3001[3836] - Attempting to avoid data races in async_traversal while evaluating dataflow()
- PR #3000[3837] - Adding hpx::util::optional as a first step to replace experimental::optional

[3815] https://github.com/STEllAR-GROUP/hpx/pull/3024
[3816] https://github.com/STEllAR-GROUP/hpx/pull/3023
[3817] https://github.com/STEllAR-GROUP/hpx/pull/3022
[3818] https://github.com/STEllAR-GROUP/hpx/pull/3021
[3819] https://github.com/STEllAR-GROUP/hpx/pull/3020
[3820] https://github.com/STEllAR-GROUP/hpx/pull/3019
[3821] https://github.com/STEllAR-GROUP/hpx/pull/3018
[3822] https://github.com/STEllAR-GROUP/hpx/pull/3017
[3823] https://github.com/STEllAR-GROUP/hpx/pull/3016
[3824] https://github.com/STEllAR-GROUP/hpx/pull/3015
[3825] https://github.com/STEllAR-GROUP/hpx/issues/3014
[3826] https://github.com/STEllAR-GROUP/hpx/issues/3013
[3827] https://github.com/STEllAR-GROUP/hpx/pull/3012
[3828] https://github.com/STEllAR-GROUP/hpx/pull/3011
[3829] https://github.com/STEllAR-GROUP/hpx/pull/3010
[3830] https://github.com/STEllAR-GROUP/hpx/pull/3009
[3831] https://github.com/STEllAR-GROUP/hpx/pull/3008
[3832] https://github.com/STEllAR-GROUP/hpx/pull/3007
[3833] https://github.com/STEllAR-GROUP/hpx/issues/3006
[3834] https://github.com/STEllAR-GROUP/hpx/pull/3004
[3835] https://github.com/STEllAR-GROUP/hpx/issues/3003
[3836] https://github.com/STEllAR-GROUP/hpx/pull/3001
[3837] https://github.com/STEllAR-GROUP/hpx/pull/3000

- PR #2998[3838] - Cleanup up and Fixing component creation and deletion
- Issue #2996[3839] - Build fails with HPX_WITH_HWLOC=OFF
- PR #2995[3840] - Push more future_data functionality to source file
- PR #2994[3841] - WIP: Fix throttle test
- PR #2993[3842] - Making sure –hpx:help does not throw for required (but missing) arguments
- PR #2992[3843] - Adding non-blocking (on destruction) service executors
- Issue #2991[3844] - run_as_os_thread locks up
- Issue #2990[3845] - –help will not work until all required options are provided
- PR #2989[3846] - Improve error messages caused by misuse of dataflow
- PR #2988[3847] - Improve error messages caused by misuse of .then
- Issue #2987[3848] - stack overflow detection producing false positives
- PR #2986[3849] - Deduplicate non-dependent thread_info logging types
- PR #2985[3850] - Adapted parallel::{all_of|any_of|none_of} for Ranges TS (see #1668)
- PR #2984[3851] - Refactor one_size_heap code to simplify code
- PR #2983[3852] - Fixing local_new_component
- PR #2982[3853] - Clang tidy
- PR #2981[3854] - Simplify allocator rebinding in pack traversal
- PR #2979[3855] - Fixing integer overflows
- PR #2978[3856] - Implement parallel::inplace_merge
- Issue #2977[3857] - Make hwloc compulsory instead of optional
- PR #2976[3858] - Making sure client_base instance that registered the component does not unregister it when being destructed
- PR #2975[3859] - Change version of pulled APEX to master
- PR #2974[3860] - Fix domain not being freed at the end of scheduling loop

3838 https://github.com/STEllAR-GROUP/hpx/pull/2998
3839 https://github.com/STEllAR-GROUP/hpx/issues/2996
3840 https://github.com/STEllAR-GROUP/hpx/pull/2995
3841 https://github.com/STEllAR-GROUP/hpx/pull/2994
3842 https://github.com/STEllAR-GROUP/hpx/pull/2993
3843 https://github.com/STEllAR-GROUP/hpx/pull/2992
3844 https://github.com/STEllAR-GROUP/hpx/issues/2991
3845 https://github.com/STEllAR-GROUP/hpx/issues/2990
3846 https://github.com/STEllAR-GROUP/hpx/pull/2989
3847 https://github.com/STEllAR-GROUP/hpx/pull/2988
3848 https://github.com/STEllAR-GROUP/hpx/issues/2987
3849 https://github.com/STEllAR-GROUP/hpx/pull/2986
3850 https://github.com/STEllAR-GROUP/hpx/pull/2985
3851 https://github.com/STEllAR-GROUP/hpx/pull/2984
3852 https://github.com/STEllAR-GROUP/hpx/pull/2983
3853 https://github.com/STEllAR-GROUP/hpx/pull/2982
3854 https://github.com/STEllAR-GROUP/hpx/pull/2981
3855 https://github.com/STEllAR-GROUP/hpx/pull/2979
3856 https://github.com/STEllAR-GROUP/hpx/pull/2978
3857 https://github.com/STEllAR-GROUP/hpx/issues/2977
3858 https://github.com/STEllAR-GROUP/hpx/pull/2976
3859 https://github.com/STEllAR-GROUP/hpx/pull/2975
3860 https://github.com/STEllAR-GROUP/hpx/pull/2974

- PR #2973[3861] - Fix small typos
- PR #2972[3862] - Adding uintstd.h header
- PR #2971[3863] - Fall back to creating local components using local_new
- PR #2970[3864] - Improve is_tuple_like trait
- PR #2969[3865] - Fix HPX_WITH_MORE_THAN_64_THREADS default value
- PR #2968[3866] - Cleaning up dataflow overload set
- PR #2967[3867] - Make parallel::merge is stable. (Fix #2964.)
- PR #2966[3868] - Fixing a couple of held locks during exception handling
- PR #2965[3869] - Adding missing #include
- Issue #2964[3870] - parallel merge is not stable
- PR #2963[3871] - Making sure any function object passed to dataflow is released after being invoked
- PR #2962[3872] - Partially reverting #2891
- PR #2961[3873] - Attempt to fix the gcc 4.9 problem with the async pack traversal
- Issue #2959[3874] - Program terminates during error handling
- Issue #2958[3875] - HPX_PLAIN_ACTION breaks due to missing include
- PR #2957[3876] - Fixing errors generated by mixing different attribute syntaxes
- Issue #2956[3877] - Mixing attribute syntaxes leads to compiler errors
- Issue #2955[3878] - Fix OS-Thread throttling
- PR #2953[3879] - Making sure any hpx.os_threads=N supplied through a –hpx::config file is taken into account
- PR #2952[3880] - Removing wrong call to cleanup_terminated_locked
- PR #2951[3881] - Revert "Make sure the function vtables are initialized before use"
- PR #2950[3882] - Fix a namespace compilation error when some schedulers are disabled
- Issue #2949[3883] - master branch giving lockups on shutdown

[3861] https://github.com/STEllAR-GROUP/hpx/pull/2973
[3862] https://github.com/STEllAR-GROUP/hpx/pull/2972
[3863] https://github.com/STEllAR-GROUP/hpx/pull/2971
[3864] https://github.com/STEllAR-GROUP/hpx/pull/2970
[3865] https://github.com/STEllAR-GROUP/hpx/pull/2969
[3866] https://github.com/STEllAR-GROUP/hpx/pull/2968
[3867] https://github.com/STEllAR-GROUP/hpx/pull/2967
[3868] https://github.com/STEllAR-GROUP/hpx/pull/2966
[3869] https://github.com/STEllAR-GROUP/hpx/pull/2965
[3870] https://github.com/STEllAR-GROUP/hpx/issues/2964
[3871] https://github.com/STEllAR-GROUP/hpx/pull/2963
[3872] https://github.com/STEllAR-GROUP/hpx/pull/2962
[3873] https://github.com/STEllAR-GROUP/hpx/pull/2961
[3874] https://github.com/STEllAR-GROUP/hpx/issues/2959
[3875] https://github.com/STEllAR-GROUP/hpx/issues/2958
[3876] https://github.com/STEllAR-GROUP/hpx/pull/2957
[3877] https://github.com/STEllAR-GROUP/hpx/issues/2956
[3878] https://github.com/STEllAR-GROUP/hpx/issues/2955
[3879] https://github.com/STEllAR-GROUP/hpx/pull/2953
[3880] https://github.com/STEllAR-GROUP/hpx/pull/2952
[3881] https://github.com/STEllAR-GROUP/hpx/pull/2951
[3882] https://github.com/STEllAR-GROUP/hpx/pull/2950
[3883] https://github.com/STEllAR-GROUP/hpx/issues/2949

- Issue #2947[3884] - hpx.ini is not used correctly at initialization
- PR #2946[3885] - Adding explicit feature test for thread_local
- PR #2945[3886] - Make sure the function vtables are initialized before use
- PR #2944[3887] - Attempting to solve affinity problems on CircleCI
- PR #2943[3888] - Changing channel actions to be direct
- PR #2942[3889] - Adding split_future for std::vector
- PR #2941[3890] - Add a feature test to test for CXX11 override
- Issue #2940[3891] - Add split_future for future<vector<T>>
- PR #2939[3892] - Making error reporting during problems with setting affinity masks more verbose
- PR #2938[3893] - Fix this various executors
- PR #2937[3894] - Fix some typos in documentation
- PR #2934[3895] - Remove the need for "complete" SFINAE checks
- PR #2933[3896] - Making sure parallel::for_loop is executed in parallel if requested
- PR #2932[3897] - Classify chunk_size_iterator to input iterator tag. (Fix #2866)
- Issue #2931[3898] - –hpx:help triggers unusual error with clang build
- PR #2930[3899] - Add #include files needed to set _POSIX_VERSION for debug check
- PR #2929[3900] - Fix a couple of deprecated c++ features
- PR #2928[3901] - Fixing execution parameters
- Issue #2927[3902] - CMake warning: … cycle in constraint graph
- PR #2926[3903] - Default pool rename
- Issue #2925[3904] - Default pool cannot be renamed
- Issue #2924[3905] - hpx:attach-debugger=startup does not work any more
- PR #2923[3906] - Alloc membind

[3884] https://github.com/STEllAR-GROUP/hpx/issues/2947
[3885] https://github.com/STEllAR-GROUP/hpx/pull/2946
[3886] https://github.com/STEllAR-GROUP/hpx/pull/2945
[3887] https://github.com/STEllAR-GROUP/hpx/pull/2944
[3888] https://github.com/STEllAR-GROUP/hpx/pull/2943
[3889] https://github.com/STEllAR-GROUP/hpx/pull/2942
[3890] https://github.com/STEllAR-GROUP/hpx/pull/2941
[3891] https://github.com/STEllAR-GROUP/hpx/issues/2940
[3892] https://github.com/STEllAR-GROUP/hpx/pull/2939
[3893] https://github.com/STEllAR-GROUP/hpx/pull/2938
[3894] https://github.com/STEllAR-GROUP/hpx/pull/2937
[3895] https://github.com/STEllAR-GROUP/hpx/pull/2934
[3896] https://github.com/STEllAR-GROUP/hpx/pull/2933
[3897] https://github.com/STEllAR-GROUP/hpx/pull/2932
[3898] https://github.com/STEllAR-GROUP/hpx/issues/2931
[3899] https://github.com/STEllAR-GROUP/hpx/pull/2930
[3900] https://github.com/STEllAR-GROUP/hpx/pull/2929
[3901] https://github.com/STEllAR-GROUP/hpx/pull/2928
[3902] https://github.com/STEllAR-GROUP/hpx/issues/2927
[3903] https://github.com/STEllAR-GROUP/hpx/pull/2926
[3904] https://github.com/STEllAR-GROUP/hpx/issues/2925
[3905] https://github.com/STEllAR-GROUP/hpx/issues/2924
[3906] https://github.com/STEllAR-GROUP/hpx/pull/2923

- PR #2922[3907] - This fixes CircleCI errors when running with –hpx:bind=none
- PR #2921[3908] - Custom pool executor was missing priority and stacksize options
- PR #2920[3909] - Adding test to trigger problem reported in #2916
- PR #2919[3910] - Make sure the resource_partitioner is properly destructed on hpx::finalize
- Issue #2918[3911] - hpx::init calls wrong (first) callback when called multiple times
- PR #2917[3912] - Adding util::checkpoint
- Issue #2916[3913] - Weird runtime failures when using a channel and chained continuations
- PR #2915[3914] - Introduce executor parameters customization points
- Issue #2914[3915] - Task assignment to current Pool has unintended consequences
- PR #2913[3916] - Fix rp hang
- PR #2912[3917] - Update contributors
- PR #2911[3918] - Fixing CUDA problems
- PR #2910[3919] - Improve error reporting for process component on POSIX systems
- PR #2909[3920] - Fix typo in include path
- PR #2908[3921] - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2907[3922] - Optionally force-delete remaining channel items on close
- PR #2906[3923] - Making sure generated performance counter names are correct
- Issue #2905[3924] - collecting idle-rate performance counters on multiple localities produces an error
- Issue #2904[3925] - build broken for Intel 17 compilers
- PR #2903[3926] - Documentation Updates– Adding New People
- PR #2902[3927] - Fixing service_executor
- PR #2901[3928] - Fixing partitioned_vector creation
- PR #2900[3929] - Add numa-balanced mode to hpx::bind, spread cores over numa domains

[3907] https://github.com/STEllAR-GROUP/hpx/pull/2922
[3908] https://github.com/STEllAR-GROUP/hpx/pull/2921
[3909] https://github.com/STEllAR-GROUP/hpx/pull/2920
[3910] https://github.com/STEllAR-GROUP/hpx/pull/2919
[3911] https://github.com/STEllAR-GROUP/hpx/issues/2918
[3912] https://github.com/STEllAR-GROUP/hpx/pull/2917
[3913] https://github.com/STEllAR-GROUP/hpx/issues/2916
[3914] https://github.com/STEllAR-GROUP/hpx/pull/2915
[3915] https://github.com/STEllAR-GROUP/hpx/issues/2914
[3916] https://github.com/STEllAR-GROUP/hpx/pull/2913
[3917] https://github.com/STEllAR-GROUP/hpx/pull/2912
[3918] https://github.com/STEllAR-GROUP/hpx/pull/2911
[3919] https://github.com/STEllAR-GROUP/hpx/pull/2910
[3920] https://github.com/STEllAR-GROUP/hpx/pull/2909
[3921] https://github.com/STEllAR-GROUP/hpx/pull/2908
[3922] https://github.com/STEllAR-GROUP/hpx/pull/2907
[3923] https://github.com/STEllAR-GROUP/hpx/pull/2906
[3924] https://github.com/STEllAR-GROUP/hpx/issues/2905
[3925] https://github.com/STEllAR-GROUP/hpx/issues/2904
[3926] https://github.com/STEllAR-GROUP/hpx/pull/2903
[3927] https://github.com/STEllAR-GROUP/hpx/pull/2902
[3928] https://github.com/STEllAR-GROUP/hpx/pull/2901
[3929] https://github.com/STEllAR-GROUP/hpx/pull/2900

- Issue #2899[3930] - hpx::bind does not have a mode that balances cores over numa domains
- PR #2898[3931] - Adding missing #include and missing guard for optional code section
- PR #2897[3932] - Removing dependency on Boost.ICL
- Issue #2896[3933] - Debug build fails without -fpermissive with GCC 7.1 and Boost 1.65
- PR #2895[3934] - Fixing SLURM environment parsing
- PR #2894[3935] - Fix incorrect handling of compile definition with value 0
- Issue #2893[3936] - Disabling schedulers causes build errors
- PR #2892[3937] - added list serializer
- PR #2891[3938] - Resource Partitioner Fixes
- Issue #2890[3939] - Destroying a non-empty channel causes an assertion failure
- PR #2889[3940] - Add check for libatomic
- PR #2888[3941] - Fix compilation problems if HPX_WITH_ITT_NOTIFY=ON
- PR #2887[3942] - Adapt broadcast() to non-unwrapping async<Action>
- PR #2886[3943] - Replace Boost.Random with C++11 <random>
- Issue #2885[3944] - regression in broadcast?
- Issue #2884[3945] - linking `-latomic` is not portable
- PR #2883[3946] - Explicitly set -pthread flag if available
- PR #2882[3947] - Wrap boost::format uses
- Issue #2881[3948] - hpx not compiling with `HPX_WITH_ITTNOTIFY=On`
- Issue #2880[3949] - hpx::bind scatter/balanced give wrong pu masks
- PR #2878[3950] - Fix incorrect pool usage masks setup in RP/thread manager
- PR #2877[3951] - Require `std::array` by default
- PR #2875[3952] - Deprecate use of BOOST_ASSERT

[3930] https://github.com/STEllAR-GROUP/hpx/issues/2899
[3931] https://github.com/STEllAR-GROUP/hpx/pull/2898
[3932] https://github.com/STEllAR-GROUP/hpx/pull/2897
[3933] https://github.com/STEllAR-GROUP/hpx/issues/2896
[3934] https://github.com/STEllAR-GROUP/hpx/pull/2895
[3935] https://github.com/STEllAR-GROUP/hpx/pull/2894
[3936] https://github.com/STEllAR-GROUP/hpx/issues/2893
[3937] https://github.com/STEllAR-GROUP/hpx/pull/2892
[3938] https://github.com/STEllAR-GROUP/hpx/pull/2891
[3939] https://github.com/STEllAR-GROUP/hpx/issues/2890
[3940] https://github.com/STEllAR-GROUP/hpx/pull/2889
[3941] https://github.com/STEllAR-GROUP/hpx/pull/2888
[3942] https://github.com/STEllAR-GROUP/hpx/pull/2887
[3943] https://github.com/STEllAR-GROUP/hpx/pull/2886
[3944] https://github.com/STEllAR-GROUP/hpx/issues/2885
[3945] https://github.com/STEllAR-GROUP/hpx/issues/2884
[3946] https://github.com/STEllAR-GROUP/hpx/pull/2883
[3947] https://github.com/STEllAR-GROUP/hpx/pull/2882
[3948] https://github.com/STEllAR-GROUP/hpx/issues/2881
[3949] https://github.com/STEllAR-GROUP/hpx/issues/2880
[3950] https://github.com/STEllAR-GROUP/hpx/pull/2878
[3951] https://github.com/STEllAR-GROUP/hpx/pull/2877
[3952] https://github.com/STEllAR-GROUP/hpx/pull/2875

- PR #2874[3953] - Changed serialization of boost.variant to use variadic templates
- Issue #2873[3954] - building with parcelport_mpi fails on cori
- PR #2871[3955] - Adding missing support for throttling scheduler
- PR #2870[3956] - Disambiguate use of base_lco_with_value macros with channel
- Issue #2869[3957] - Difficulty compiling `HPX_REGISTER_CHANNEL_DECLARATION(double)`
- PR #2868[3958] - Removing unneeded assert
- PR #2867[3959] - Implement parallel::unique
- Issue #2866[3960] - The chunk_size_iterator violates multipass guarantee
- PR #2865[3961] - Only use sched_getcpu on linux machines
- PR #2864[3962] - Create redistribution archive for successful builds
- PR #2863[3963] - Replace casts/assignments with hard-coded memcpy operations
- Issue #2862[3964] - sched_getcpu not available on MacOS
- PR #2861[3965] - Fixing unmatched header defines and recursive inclusion of threadmanager
- Issue #2860[3966] - Master program fails with assertion 'type == data_type_address' failed: HPX(assertion_failure)
- Issue #2852[3967] - Support for ARM64
- PR #2858[3968] - Fix misplaced #if #endif's that cause build failure without THREAD_CUMULATIVE_COUNTS
- PR #2857[3969] - Fix some listing in documentation
- PR #2856[3970] - Fixing component handling for lcos
- PR #2855[3971] - Add documentation for coarrays
- PR #2854[3972] - Support ARM64 in timestamps
- PR #2853[3973] - Update Table 17. Non-modifying Parallel Algorithms in Documentation
- PR #2851[3974] - Allowing for non-default-constructible component types
- PR #2850[3975] - Enable returning future<R> from actions where R is not default-constructible

3953 https://github.com/STEllAR-GROUP/hpx/pull/2874
3954 https://github.com/STEllAR-GROUP/hpx/issues/2873
3955 https://github.com/STEllAR-GROUP/hpx/pull/2871
3956 https://github.com/STEllAR-GROUP/hpx/pull/2870
3957 https://github.com/STEllAR-GROUP/hpx/issues/2869
3958 https://github.com/STEllAR-GROUP/hpx/pull/2868
3959 https://github.com/STEllAR-GROUP/hpx/pull/2867
3960 https://github.com/STEllAR-GROUP/hpx/issues/2866
3961 https://github.com/STEllAR-GROUP/hpx/pull/2865
3962 https://github.com/STEllAR-GROUP/hpx/pull/2864
3963 https://github.com/STEllAR-GROUP/hpx/pull/2863
3964 https://github.com/STEllAR-GROUP/hpx/issues/2862
3965 https://github.com/STEllAR-GROUP/hpx/pull/2861
3966 https://github.com/STEllAR-GROUP/hpx/issues/2860
3967 https://github.com/STEllAR-GROUP/hpx/issues/2852
3968 https://github.com/STEllAR-GROUP/hpx/pull/2858
3969 https://github.com/STEllAR-GROUP/hpx/pull/2857
3970 https://github.com/STEllAR-GROUP/hpx/pull/2856
3971 https://github.com/STEllAR-GROUP/hpx/pull/2855
3972 https://github.com/STEllAR-GROUP/hpx/pull/2854
3973 https://github.com/STEllAR-GROUP/hpx/pull/2853
3974 https://github.com/STEllAR-GROUP/hpx/pull/2851
3975 https://github.com/STEllAR-GROUP/hpx/pull/2850

- PR #2849[3976] - Unify serialization of non-default-constructable types
- Issue #2848[3977] - Components have to be default constructible
- Issue #2847[3978] - Returning a future<R> where R is not default-constructable broken
- Issue #2846[3979] - Unify serialization of non-default-constructible types
- PR #2845[3980] - Add Visual Studio 2015 to the tested toolchains in Appveyor
- Issue #2844[3981] - Change the appveyor build to use the minimal required MSVC version
- Issue #2843[3982] - multi node hello_world hangs
- PR #2842[3983] - Correcting Spelling mistake in docs
- PR #2841[3984] - Fix usage of std::aligned_storage
- PR #2840[3985] - Remove constexpr from a void function
- Issue #2839[3986] - memcpy buffer overflow: load_construct_data() and std::complex members
- Issue #2835[3987] - `constexpr` functions with `void` return type break compilation with CUDA 8.0
- Issue #2834[3988] - One suspicion in parallel::detail::handle_exception
- PR #2833[3989] - Implement parallel::merge
- PR #2832[3990] - Fix a strange thing in parallel::util::detail::handle_local_exceptions. (Fix #2818)
- PR #2830[3991] - Break the debugger when a test failed
- Issue #2831[3992] - `parallel/executors/execution_fwd.hpp` causes compilation failure in C++11 mode.
- PR #2829[3993] - Implement an API for asynchronous pack traversal
- PR #2828[3994] - Split unit test builds on CircleCI to avoid timeouts
- Issue #2827[3995] - failure to compile hello_world example with -Werror
- PR #2824[3996] - Making sure promises are marked as started when used as continuations
- PR #2823[3997] - Add documentation for partitioned_vector_view
- Issue #2822[3998] - Yet another issue with wait_for similar to #2796

3976 https://github.com/STEllAR-GROUP/hpx/pull/2849
3977 https://github.com/STEllAR-GROUP/hpx/issues/2848
3978 https://github.com/STEllAR-GROUP/hpx/issues/2847
3979 https://github.com/STEllAR-GROUP/hpx/issues/2846
3980 https://github.com/STEllAR-GROUP/hpx/pull/2845
3981 https://github.com/STEllAR-GROUP/hpx/issues/2844
3982 https://github.com/STEllAR-GROUP/hpx/issues/2843
3983 https://github.com/STEllAR-GROUP/hpx/pull/2842
3984 https://github.com/STEllAR-GROUP/hpx/pull/2841
3985 https://github.com/STEllAR-GROUP/hpx/pull/2840
3986 https://github.com/STEllAR-GROUP/hpx/issues/2839
3987 https://github.com/STEllAR-GROUP/hpx/issues/2835
3988 https://github.com/STEllAR-GROUP/hpx/issues/2834
3989 https://github.com/STEllAR-GROUP/hpx/pull/2833
3990 https://github.com/STEllAR-GROUP/hpx/pull/2832
3991 https://github.com/STEllAR-GROUP/hpx/pull/2830
3992 https://github.com/STEllAR-GROUP/hpx/issues/2831
3993 https://github.com/STEllAR-GROUP/hpx/pull/2829
3994 https://github.com/STEllAR-GROUP/hpx/pull/2828
3995 https://github.com/STEllAR-GROUP/hpx/issues/2827
3996 https://github.com/STEllAR-GROUP/hpx/pull/2824
3997 https://github.com/STEllAR-GROUP/hpx/pull/2823
3998 https://github.com/STEllAR-GROUP/hpx/issues/2822

- PR #2821[3999] - Fix bugs and improve that about HPX_HAVE_CXX11_AUTO_RETURN_VALUE of CMake
- PR #2820[4000] - Support C++11 in benchmark codes of parallel::partition and parallel::partition_copy
- PR #2819[4001] - Fix compile errors in unit test of container version of parallel::partition
- Issue #2818[4002] - A strange thing in parallel::util::detail::handle_local_exceptions
- Issue #2815[4003] - HPX fails to compile with HPX_WITH_CUDA=ON and the new CUDA 9.0 RC
- Issue #2814[4004] - Using 'gmakeN' after 'cmake' produces error in src/CMakeFiles/hpx.dir/runtime/agas/addressing_service.cpp.o
- PR #2813[4005] - Properly support [[noreturn]] attribute if available
- Issue #2812[4006] - Compilation fails with gcc 7.1.1
- PR #2811[4007] - Adding hpx::launch::lazy and support for async, dataflow, and future::then
- PR #2810[4008] - Add option allowing to disable deprecation warning
- PR #2809[4009] - Disable throttling scheduler if HWLOC is not found/used
- PR #2808[4010] - Fix compile errors on some environments of parallel::partition
- Issue #2807[4011] - Difficulty building with `HPX_WITH_HWLOC=Off`
- PR #2806[4012] - Partitioned vector
- PR #2805[4013] - Serializing collections with non-default constructible data
- PR #2802[4014] - Fix FreeBSD 11
- Issue #2801[4015] - Rate limiting techniques in io_service
- Issue #2800[4016] - New Launch Policy: async_if
- PR #2799[4017] - Fix a unit test failure on GCC in tuple_cat
- PR #2798[4018] - bump minimum required cmake to 3.0 in test
- PR #2797[4019] - Making sure future::wait_for et.al. work properly for action results
- Issue #2796[4020] - wait_for does always in "deferred" state for calls on remote localities
- Issue #2795[4021] - Serialization of types without default constructor

[3999] https://github.com/STEllAR-GROUP/hpx/pull/2821
[4000] https://github.com/STEllAR-GROUP/hpx/pull/2820
[4001] https://github.com/STEllAR-GROUP/hpx/pull/2819
[4002] https://github.com/STEllAR-GROUP/hpx/issues/2818
[4003] https://github.com/STEllAR-GROUP/hpx/issues/2815
[4004] https://github.com/STEllAR-GROUP/hpx/issues/2814
[4005] https://github.com/STEllAR-GROUP/hpx/pull/2813
[4006] https://github.com/STEllAR-GROUP/hpx/issues/2812
[4007] https://github.com/STEllAR-GROUP/hpx/pull/2811
[4008] https://github.com/STEllAR-GROUP/hpx/pull/2810
[4009] https://github.com/STEllAR-GROUP/hpx/pull/2809
[4010] https://github.com/STEllAR-GROUP/hpx/pull/2808
[4011] https://github.com/STEllAR-GROUP/hpx/issues/2807
[4012] https://github.com/STEllAR-GROUP/hpx/pull/2806
[4013] https://github.com/STEllAR-GROUP/hpx/pull/2805
[4014] https://github.com/STEllAR-GROUP/hpx/pull/2802
[4015] https://github.com/STEllAR-GROUP/hpx/issues/2801
[4016] https://github.com/STEllAR-GROUP/hpx/issues/2800
[4017] https://github.com/STEllAR-GROUP/hpx/pull/2799
[4018] https://github.com/STEllAR-GROUP/hpx/pull/2798
[4019] https://github.com/STEllAR-GROUP/hpx/pull/2797
[4020] https://github.com/STEllAR-GROUP/hpx/issues/2796
[4021] https://github.com/STEllAR-GROUP/hpx/issues/2795

- PR #2794[4022] - Fixing test for partitioned_vector iteration
- PR #2792[4023] - Implemented segmented find and its variations for partitioned vector
- PR #2791[4024] - Circumvent scary warning about placement new
- PR #2790[4025] - Fix OSX build
- PR #2789[4026] - Resource partitioner
- PR #2788[4027] - Adapt parallel::is_heap and parallel::is_heap_until to Ranges TS
- PR #2787[4028] - Unwrap hotfixes
- PR #2786[4029] - Update CMake Minimum Version to 3.3.2 (refs #2565)
- Issue #2785[4030] - Issues with masks and cpuset
- PR #2784[4031] - Error with reduce and transform reduce fixed
- PR #2783[4032] - StackOverflow integration with libsigsegv
- PR #2782[4033] - Replace boost::atomic with std::atomic (where possible)
- PR #2781[4034] - Check for and optionally use [[deprecated]] attribute
- PR #2780[4035] - Adding empty (but non-trivial) destructor to circumvent warnings
- PR #2779[4036] - Exception info tweaks
- PR #2778[4037] - Implement parallel::partition
- PR #2777[4038] - Improve error handling in gather_here/gather_there
- PR #2776[4039] - Fix a bug in compiler version check
- PR #2775[4040] - Fix compilation when HPX_WITH_LOGGING is OFF
- PR #2774[4041] - Removing dependency on Boost.Date_Time
- PR #2773[4042] - Add sync_images() method to spmd_block class
- PR #2772[4043] - Adding documentation for PAPI counters
- PR #2771[4044] - Removing boost preprocessor dependency

[4022] https://github.com/STEllAR-GROUP/hpx/pull/2794
[4023] https://github.com/STEllAR-GROUP/hpx/pull/2792
[4024] https://github.com/STEllAR-GROUP/hpx/pull/2791
[4025] https://github.com/STEllAR-GROUP/hpx/pull/2790
[4026] https://github.com/STEllAR-GROUP/hpx/pull/2789
[4027] https://github.com/STEllAR-GROUP/hpx/pull/2788
[4028] https://github.com/STEllAR-GROUP/hpx/pull/2787
[4029] https://github.com/STEllAR-GROUP/hpx/pull/2786
[4030] https://github.com/STEllAR-GROUP/hpx/issues/2785
[4031] https://github.com/STEllAR-GROUP/hpx/pull/2784
[4032] https://github.com/STEllAR-GROUP/hpx/pull/2783
[4033] https://github.com/STEllAR-GROUP/hpx/pull/2782
[4034] https://github.com/STEllAR-GROUP/hpx/pull/2781
[4035] https://github.com/STEllAR-GROUP/hpx/pull/2780
[4036] https://github.com/STEllAR-GROUP/hpx/pull/2779
[4037] https://github.com/STEllAR-GROUP/hpx/pull/2778
[4038] https://github.com/STEllAR-GROUP/hpx/pull/2777
[4039] https://github.com/STEllAR-GROUP/hpx/pull/2776
[4040] https://github.com/STEllAR-GROUP/hpx/pull/2775
[4041] https://github.com/STEllAR-GROUP/hpx/pull/2774
[4042] https://github.com/STEllAR-GROUP/hpx/pull/2773
[4043] https://github.com/STEllAR-GROUP/hpx/pull/2772
[4044] https://github.com/STEllAR-GROUP/hpx/pull/2771

- PR #2770[4045] - Adding test, fixing deadlock in config registry
- PR #2769[4046] - Remove some other warnings and errors detected by clang 5.0
- Issue #2768[4047] - Is there iterator tag for HPX?
- PR #2767[4048] - Improvements to continuation annotation
- PR #2765[4049] - gcc split stack support for HPX threads #620
- PR #2764[4050] - Fix some uses of begin/end, remove unnecessary includes
- PR #2763[4051] - Bump minimal Boost version to 1.55.0
- PR #2762[4052] - hpx::partitioned_vector serializer
- PR #2761[4053] - Adding configuration summary to cmake output and –hpx:info
- PR #2760[4054] - Removing 1d_hydro example as it is broken
- PR #2758[4055] - Remove various warnings detected by clang 5.0
- Issue #2757[4056] - In case of a "raw thread" is needed per core for implementing parallel algorithm, what is good practice in HPX?
- PR #2756[4057] - Allowing for LCOs to be simple components
- PR #2755[4058] - Removing make_index_pack_unrolled
- PR #2754[4059] - Implement parallel::unique_copy
- PR #2753[4060] - Fixing detection of [[fallthrough]] attribute
- PR #2752[4061] - New thread priority names
- PR #2751[4062] - Replace boost::exception with proposed exception_info
- PR #2750[4063] - Replace boost::iterator_range
- PR #2749[4064] - Fixing hdf5 examples
- Issue #2748[4065] - HPX fails to build with enabled hdf5 examples
- Issue #2747[4066] - Inherited task priorities break certain DAG optimizations
- Issue #2746[4067] - HPX segfaulting with valgrind

[4045] https://github.com/STEllAR-GROUP/hpx/pull/2770
[4046] https://github.com/STEllAR-GROUP/hpx/pull/2769
[4047] https://github.com/STEllAR-GROUP/hpx/issues/2768
[4048] https://github.com/STEllAR-GROUP/hpx/pull/2767
[4049] https://github.com/STEllAR-GROUP/hpx/pull/2765
[4050] https://github.com/STEllAR-GROUP/hpx/pull/2764
[4051] https://github.com/STEllAR-GROUP/hpx/pull/2763
[4052] https://github.com/STEllAR-GROUP/hpx/pull/2762
[4053] https://github.com/STEllAR-GROUP/hpx/pull/2761
[4054] https://github.com/STEllAR-GROUP/hpx/pull/2760
[4055] https://github.com/STEllAR-GROUP/hpx/pull/2758
[4056] https://github.com/STEllAR-GROUP/hpx/issues/2757
[4057] https://github.com/STEllAR-GROUP/hpx/pull/2756
[4058] https://github.com/STEllAR-GROUP/hpx/pull/2755
[4059] https://github.com/STEllAR-GROUP/hpx/pull/2754
[4060] https://github.com/STEllAR-GROUP/hpx/pull/2753
[4061] https://github.com/STEllAR-GROUP/hpx/pull/2752
[4062] https://github.com/STEllAR-GROUP/hpx/pull/2751
[4063] https://github.com/STEllAR-GROUP/hpx/pull/2750
[4064] https://github.com/STEllAR-GROUP/hpx/pull/2749
[4065] https://github.com/STEllAR-GROUP/hpx/issues/2748
[4066] https://github.com/STEllAR-GROUP/hpx/issues/2747
[4067] https://github.com/STEllAR-GROUP/hpx/issues/2746

- PR #2745[4068] - Adding extended arithmetic performance counters
- PR #2744[4069] - Adding ability to statistics counters to reset base counter
- Issue #2743[4070] - Statistics counter does not support resetting
- PR #2742[4071] - Making sure Vc V2 builds without additional HPX configuration flags
- PR #2741[4072] - Deprecate unwrapped and implement unwrap and unwrapping
- PR #2740[4073] - Coroutine stackoverflow detection for linux/posix; Issue #2408
- PR #2739[4074] - Add files via upload
- PR #2738[4075] - Appveyor support
- PR #2737[4076] - Fixing 2735
- Issue #2736[4077] - 1d_hydro example doesn't work
- Issue #2735[4078] - partitioned_vector_subview test failing
- PR #2734[4079] - Add C++11 range utilities
- PR #2733[4080] - Adapting iterator requirements for parallel algorithms
- PR #2732[4081] - Integrate C++ Co-arrays
- PR #2731[4082] - Adding on_migrated event handler to migratable component instances
- Issue #2729[4083] - Add on_migrated() event handler to migratable components
- Issue #2728[4084] - Why Projection is needed in parallel algorithms?
- PR #2727[4085] - Cmake files for StackOverflow Detection
- PR #2726[4086] - CMake for Stack Overflow Detection
- PR #2725[4087] - Implemented segmented algorithms for partitioned vector
- PR #2724[4088] - Fix examples in Action documentation
- PR #2723[4089] - Enable lcos::channel<T>::register_as
- Issue #2722[4090] - channel register_as() failing on compilation

[4068] https://github.com/STEllAR-GROUP/hpx/pull/2745
[4069] https://github.com/STEllAR-GROUP/hpx/pull/2744
[4070] https://github.com/STEllAR-GROUP/hpx/issues/2743
[4071] https://github.com/STEllAR-GROUP/hpx/pull/2742
[4072] https://github.com/STEllAR-GROUP/hpx/pull/2741
[4073] https://github.com/STEllAR-GROUP/hpx/pull/2740
[4074] https://github.com/STEllAR-GROUP/hpx/pull/2739
[4075] https://github.com/STEllAR-GROUP/hpx/pull/2738
[4076] https://github.com/STEllAR-GROUP/hpx/pull/2737
[4077] https://github.com/STEllAR-GROUP/hpx/issues/2736
[4078] https://github.com/STEllAR-GROUP/hpx/issues/2735
[4079] https://github.com/STEllAR-GROUP/hpx/pull/2734
[4080] https://github.com/STEllAR-GROUP/hpx/pull/2733
[4081] https://github.com/STEllAR-GROUP/hpx/pull/2732
[4082] https://github.com/STEllAR-GROUP/hpx/pull/2731
[4083] https://github.com/STEllAR-GROUP/hpx/issues/2729
[4084] https://github.com/STEllAR-GROUP/hpx/issues/2728
[4085] https://github.com/STEllAR-GROUP/hpx/pull/2727
[4086] https://github.com/STEllAR-GROUP/hpx/pull/2726
[4087] https://github.com/STEllAR-GROUP/hpx/pull/2725
[4088] https://github.com/STEllAR-GROUP/hpx/pull/2724
[4089] https://github.com/STEllAR-GROUP/hpx/pull/2723
[4090] https://github.com/STEllAR-GROUP/hpx/issues/2722

- PR #2721[4091] - Mind map
- PR #2720[4092] - reorder forward declarations to get rid of C++14-only auto return types
- PR #2719[4093] - Add documentation for partitioned_vector and add features in pack.hpp
- Issue #2718[4094] - Some forward declarations in execution_fwd.hpp aren't C++11-compatible
- PR #2717[4095] - Config support for fallthrough attribute
- PR #2716[4096] - Implement parallel::partition_copy
- PR #2715[4097] - initial import of icu string serializer
- PR #2714[4098] - initial import of valarray serializer
- PR #2713[4099] - Remove slashes before CMAKE_FILES_DIRECTORY variables
- PR #2712[4100] - Fixing wait for 1751
- PR #2711[4101] - Adjust code for minimal supported GCC having being bumped to 4.9
- PR #2710[4102] - Adding code of conduct
- PR #2709[4103] - Fixing UB in destroy tests
- PR #2708[4104] - Add inline to prevent multiple definition issue
- Issue #2707[4105] - Multiple defined symbols for task_block.hpp in VS2015
- PR #2706[4106] - Adding .clang-format file
- PR #2704[4107] - Add a synchronous mapping API
- Issue #2703[4108] - Request: Add the .clang-format file to the repository
- Issue #2702[4109] - STEllAR-GROUP/Vc slower than VCv1 possibly due to wrong instructions generated
- Issue #2701[4110] - Datapar with STEllAR-GROUP/Vc requires obscure flag
- Issue #2700[4111] - Naming inconsistency in parallel algorithms
- Issue #2699[4112] - Iterator requirements are different from standard in parallel copy_if.
- PR #2698[4113] - Properly releasing parcelport write handlers

[4091] https://github.com/STEllAR-GROUP/hpx/pull/2721
[4092] https://github.com/STEllAR-GROUP/hpx/pull/2720
[4093] https://github.com/STEllAR-GROUP/hpx/pull/2719
[4094] https://github.com/STEllAR-GROUP/hpx/issues/2718
[4095] https://github.com/STEllAR-GROUP/hpx/pull/2717
[4096] https://github.com/STEllAR-GROUP/hpx/pull/2716
[4097] https://github.com/STEllAR-GROUP/hpx/pull/2715
[4098] https://github.com/STEllAR-GROUP/hpx/pull/2714
[4099] https://github.com/STEllAR-GROUP/hpx/pull/2713
[4100] https://github.com/STEllAR-GROUP/hpx/pull/2712
[4101] https://github.com/STEllAR-GROUP/hpx/pull/2711
[4102] https://github.com/STEllAR-GROUP/hpx/pull/2710
[4103] https://github.com/STEllAR-GROUP/hpx/pull/2709
[4104] https://github.com/STEllAR-GROUP/hpx/pull/2708
[4105] https://github.com/STEllAR-GROUP/hpx/issues/2707
[4106] https://github.com/STEllAR-GROUP/hpx/pull/2706
[4107] https://github.com/STEllAR-GROUP/hpx/pull/2704
[4108] https://github.com/STEllAR-GROUP/hpx/issues/2703
[4109] https://github.com/STEllAR-GROUP/hpx/issues/2702
[4110] https://github.com/STEllAR-GROUP/hpx/issues/2701
[4111] https://github.com/STEllAR-GROUP/hpx/issues/2700
[4112] https://github.com/STEllAR-GROUP/hpx/issues/2699
[4113] https://github.com/STEllAR-GROUP/hpx/pull/2698

- Issue #2697[4114] - Compile error in addressing_service.cpp
- Issue #2696[4115] - Building and using HPX statically: undefined references from runtime_support_server.cpp
- Issue #2695[4116] - Executor changes cause compilation failures
- PR #2694[4117] - Refining C++ language mode detection for MSVC
- PR #2693[4118] - P0443 r2
- PR #2692[4119] - Partially reverting changes to parcel_await
- Issue #2689[4120] - HPX build fails when HPX_WITH_CUDA is enabled
- PR #2688[4121] - Make Cuda Clang builds pass
- PR #2687[4122] - Add an is_tuple_like trait for sequenceable type detection
- PR #2686[4123] - Allowing throttling scheduler to be used without idle backoff
- PR #2685[4124] - Add support of std::array to hpx::util::tuple_size and tuple_element
- PR #2684[4125] - Adding new statistics performance counters
- PR #2683[4126] - Replace boost::exception_ptr with std::exception_ptr
- Issue #2682[4127] - HPX does not compile with HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF
- PR #2681[4128] - Attempt to fix problem in managed_component_base
- PR #2680[4129] - Fix bad size during archive creation
- Issue #2679[4130] - Mismatch between size of archive and container
- Issue #2678[4131] - In parallel algorithm, other tasks are executed to the end even if an exception occurs in any task.
- PR #2677[4132] - Adding include check for std::addressof
- PR #2676[4133] - Adding parallel::destroy and destroy_n
- PR #2675[4134] - Making sure statistics counters work as expected
- PR #2674[4135] - Turning assertions into exceptions
- PR #2673[4136] - Inhibit direct conversion from future<future<T>> –> future<void>

[4114] https://github.com/STEllAR-GROUP/hpx/issues/2697
[4115] https://github.com/STEllAR-GROUP/hpx/issues/2696
[4116] https://github.com/STEllAR-GROUP/hpx/issues/2695
[4117] https://github.com/STEllAR-GROUP/hpx/pull/2694
[4118] https://github.com/STEllAR-GROUP/hpx/pull/2693
[4119] https://github.com/STEllAR-GROUP/hpx/pull/2692
[4120] https://github.com/STEllAR-GROUP/hpx/issues/2689
[4121] https://github.com/STEllAR-GROUP/hpx/pull/2688
[4122] https://github.com/STEllAR-GROUP/hpx/pull/2687
[4123] https://github.com/STEllAR-GROUP/hpx/pull/2686
[4124] https://github.com/STEllAR-GROUP/hpx/pull/2685
[4125] https://github.com/STEllAR-GROUP/hpx/pull/2684
[4126] https://github.com/STEllAR-GROUP/hpx/pull/2683
[4127] https://github.com/STEllAR-GROUP/hpx/issues/2682
[4128] https://github.com/STEllAR-GROUP/hpx/pull/2681
[4129] https://github.com/STEllAR-GROUP/hpx/pull/2680
[4130] https://github.com/STEllAR-GROUP/hpx/issues/2679
[4131] https://github.com/STEllAR-GROUP/hpx/issues/2678
[4132] https://github.com/STEllAR-GROUP/hpx/pull/2677
[4133] https://github.com/STEllAR-GROUP/hpx/pull/2676
[4134] https://github.com/STEllAR-GROUP/hpx/pull/2675
[4135] https://github.com/STEllAR-GROUP/hpx/pull/2674
[4136] https://github.com/STEllAR-GROUP/hpx/pull/2673

- PR #2672[4137] - C++17 invoke forms
- PR #2671[4138] - Adding uninitialized_value_construct and uninitialized_value_construct_n
- PR #2670[4139] - Integrate spmd multidimensional views for partitioned_vectors
- PR #2669[4140] - Adding uninitialized_default_construct and uninitialized_default_construct_n
- PR #2668[4141] - Fixing documentation index
- Issue #2667[4142] - Ambiguity of nested hpx::future<void>'s
- Issue #2666[4143] - Statistics Performance counter is not working
- PR #2664[4144] - Adding uninitialized_move and uninitialized_move_n
- Issue #2663[4145] - Seg fault in managed_component::get_base_gid, possibly cause by util::reinitializable_static
- Issue #2662[4146] - Crash in managed_component::get_base_gid due to problem with util::reinitializable_static
- PR #2665[4147] - Hide the `detail` namespace in doxygen per default
- PR #2660[4148] - Add documentation to hpx::util::unwrapped and hpx::util::unwrapped2
- PR #2659[4149] - Improve integration with vcpkg
- PR #2658[4150] - Unify access_data trait for use in both, serialization and de-serialization
- PR #2657[4151] - Removing hpx::lcos::queue<T>
- PR #2656[4152] - Reduce MAX_TERMINATED_THREADS default, improve memory use on manycore cpus
- PR #2655[4153] - Mainteinance for emulate-deleted macros
- PR #2654[4154] - Implement parallel is_heap and is_heap_until
- PR #2653[4155] - Drop support for VS2013
- PR #2652[4156] - This patch makes sure that all parcels in a batch are properly handled
- PR #2649[4157] - Update docs (Table 18) - move transform to end
- Issue #2647[4158] - hpx::parcelset::detail::parcel_data::**has_continuation_** is uninitialized
- Issue #2644[4159] - Some .vcxproj in the HPX.sln fail to build

4137 https://github.com/STEllAR-GROUP/hpx/pull/2672
4138 https://github.com/STEllAR-GROUP/hpx/pull/2671
4139 https://github.com/STEllAR-GROUP/hpx/pull/2670
4140 https://github.com/STEllAR-GROUP/hpx/pull/2669
4141 https://github.com/STEllAR-GROUP/hpx/pull/2668
4142 https://github.com/STEllAR-GROUP/hpx/issues/2667
4143 https://github.com/STEllAR-GROUP/hpx/issues/2666
4144 https://github.com/STEllAR-GROUP/hpx/pull/2664
4145 https://github.com/STEllAR-GROUP/hpx/issues/2663
4146 https://github.com/STEllAR-GROUP/hpx/issues/2662
4147 https://github.com/STEllAR-GROUP/hpx/pull/2665
4148 https://github.com/STEllAR-GROUP/hpx/pull/2660
4149 https://github.com/STEllAR-GROUP/hpx/pull/2659
4150 https://github.com/STEllAR-GROUP/hpx/pull/2658
4151 https://github.com/STEllAR-GROUP/hpx/pull/2657
4152 https://github.com/STEllAR-GROUP/hpx/pull/2656
4153 https://github.com/STEllAR-GROUP/hpx/pull/2655
4154 https://github.com/STEllAR-GROUP/hpx/pull/2654
4155 https://github.com/STEllAR-GROUP/hpx/pull/2653
4156 https://github.com/STEllAR-GROUP/hpx/pull/2652
4157 https://github.com/STEllAR-GROUP/hpx/pull/2649
4158 https://github.com/STEllAR-GROUP/hpx/issues/2647
4159 https://github.com/STEllAR-GROUP/hpx/issues/2644

- Issue #2641[4160] - `hpx::lcos::queue` should be deprecated
- PR #2640[4161] - A new throttling policy with public APIs to suspend/resume
- PR #2639[4162] - Fix a tiny typo in tutorial.
- Issue #2638[4163] - Invalid return type 'void' of constexpr function
- PR #2636[4164] - Add and use HPX_MSVC_WARNING_PRAGMA for #pragma warning
- PR #2633[4165] - Distributed define_spmd_block
- PR #2632[4166] - Making sure container serialization uses size-compatible types
- PR #2631[4167] - Add lcos::local::one_element_channel
- PR #2629[4168] - Move unordered_map out of parcelport into hpx/concurrent
- PR #2628[4169] - Making sure that shutdown does not hang
- PR #2627[4170] - Fix serialization
- PR #2626[4171] - Generate `cmake_variables.qbk` and `cmake_toolchains.qbk` outside of the source tree
- PR #2625[4172] - Supporting -std=c++17 flag
- PR #2624[4173] - Fixing a small cmake typo
- PR #2622[4174] - Update CMake minimum required version to 3.0.2 (closes #2621)
- Issue #2621[4175] - Compiling hpx master fails with /usr/bin/ld: final link failed: Bad value
- PR #2620[4176] - Remove warnings due to some captured variables
- PR #2619[4177] - LF multiple parcels
- PR #2618[4178] - Some fixes to libfabric that didn't get caught before the merge
- PR #2617[4179] - Adding `hpx::local_new`
- PR #2616[4180] - Documentation: Extract all entities in order to autolink functions correctly
- Issue #2615[4181] - Documentation: Linking functions is broken
- PR #2614[4182] - Adding serialization for std::deque

[4160] https://github.com/STEllAR-GROUP/hpx/issues/2641
[4161] https://github.com/STEllAR-GROUP/hpx/pull/2640
[4162] https://github.com/STEllAR-GROUP/hpx/pull/2639
[4163] https://github.com/STEllAR-GROUP/hpx/issues/2638
[4164] https://github.com/STEllAR-GROUP/hpx/pull/2636
[4165] https://github.com/STEllAR-GROUP/hpx/pull/2633
[4166] https://github.com/STEllAR-GROUP/hpx/pull/2632
[4167] https://github.com/STEllAR-GROUP/hpx/pull/2631
[4168] https://github.com/STEllAR-GROUP/hpx/pull/2629
[4169] https://github.com/STEllAR-GROUP/hpx/pull/2628
[4170] https://github.com/STEllAR-GROUP/hpx/pull/2627
[4171] https://github.com/STEllAR-GROUP/hpx/pull/2626
[4172] https://github.com/STEllAR-GROUP/hpx/pull/2625
[4173] https://github.com/STEllAR-GROUP/hpx/pull/2624
[4174] https://github.com/STEllAR-GROUP/hpx/pull/2622
[4175] https://github.com/STEllAR-GROUP/hpx/issues/2621
[4176] https://github.com/STEllAR-GROUP/hpx/pull/2620
[4177] https://github.com/STEllAR-GROUP/hpx/pull/2619
[4178] https://github.com/STEllAR-GROUP/hpx/pull/2618
[4179] https://github.com/STEllAR-GROUP/hpx/pull/2617
[4180] https://github.com/STEllAR-GROUP/hpx/pull/2616
[4181] https://github.com/STEllAR-GROUP/hpx/issues/2615
[4182] https://github.com/STEllAR-GROUP/hpx/pull/2614

- PR #2613[4183] - We need to link with boost.thread and boost.chrono if we use boost.context
- PR #2612[4184] - Making sure for_loop_n(par, . . .) is actually executed in parallel
- PR #2611[4185] - Add documentation to invoke_fused and friends NFC
- PR #2610[4186] - Added reduction templates using an identity value
- PR #2608[4187] - Fixing some unused vars in inspect
- PR #2607[4188] - Fixed build for mingw
- PR #2606[4189] - Supporting generic context for boost >= 1.61
- PR #2605[4190] - Parcelport libfabric3
- PR #2604[4191] - Adding allocator support to promise and friends
- PR #2603[4192] - Barrier hang
- PR #2602[4193] - Changes to scheduler to steal from one high-priority queue
- Issue #2601[4194] - High priority tasks are not executed first
- PR #2600[4195] - Compat fixes
- PR #2599[4196] - Compatibility layer for threading support
- PR #2598[4197] - V1.1
- PR #2597[4198] - Release V1.0
- PR #2592[4199] - First attempt to introduce spmd_block in hpx
- PR #2586[4200] - local_segment in segmented_iterator_traits
- Issue #2584[4201] - Add allocator support to promise, packaged_task and friends
- PR #2576[4202] - Add missing dependencies of cuda based tests
- PR #2575[4203] - Remove warnings due to some captured variables
- Issue #2574[4204] - MSVC 2015 Compiler crash when building HPX
- Issue #2568[4205] - Remove throttle_scheduler as it has been abandoned

[4183] https://github.com/STEllAR-GROUP/hpx/pull/2613
[4184] https://github.com/STEllAR-GROUP/hpx/pull/2612
[4185] https://github.com/STEllAR-GROUP/hpx/pull/2611
[4186] https://github.com/STEllAR-GROUP/hpx/pull/2610
[4187] https://github.com/STEllAR-GROUP/hpx/pull/2608
[4188] https://github.com/STEllAR-GROUP/hpx/pull/2607
[4189] https://github.com/STEllAR-GROUP/hpx/pull/2606
[4190] https://github.com/STEllAR-GROUP/hpx/pull/2605
[4191] https://github.com/STEllAR-GROUP/hpx/pull/2604
[4192] https://github.com/STEllAR-GROUP/hpx/pull/2603
[4193] https://github.com/STEllAR-GROUP/hpx/pull/2602
[4194] https://github.com/STEllAR-GROUP/hpx/issues/2601
[4195] https://github.com/STEllAR-GROUP/hpx/pull/2600
[4196] https://github.com/STEllAR-GROUP/hpx/pull/2599
[4197] https://github.com/STEllAR-GROUP/hpx/pull/2598
[4198] https://github.com/STEllAR-GROUP/hpx/pull/2597
[4199] https://github.com/STEllAR-GROUP/hpx/pull/2592
[4200] https://github.com/STEllAR-GROUP/hpx/pull/2586
[4201] https://github.com/STEllAR-GROUP/hpx/issues/2584
[4202] https://github.com/STEllAR-GROUP/hpx/pull/2576
[4203] https://github.com/STEllAR-GROUP/hpx/pull/2575
[4204] https://github.com/STEllAR-GROUP/hpx/issues/2574
[4205] https://github.com/STEllAR-GROUP/hpx/issues/2568

- Issue #2566[4206] - Add an inline versioning namespace before 1.0 release
- Issue #2565[4207] - Raise minimal cmake version requirement
- PR #2556[4208] - Fixing scan partitioner
- PR #2546[4209] - Broadcast async
- Issue #2543[4210] - make install fails due to a non-existing .so file
- PR #2495[4211] - wait_or_add_new returning thread_id_type
- Issue #2480[4212] - Unable to register new performance counter
- Issue #2471[4213] - no type named 'fcontext_t' in namespace
- Issue #2456[4214] - Re-implement hpx::util::unwrapped
- Issue #2455[4215] - Add more arithmetic performance counters
- PR #2454[4216] - Fix a couple of warnings and compiler errors
- PR #2453[4217] - Timed executor support
- PR #2447[4218] - Implementing new executor API (P0443)
- Issue #2439[4219] - Implement executor proposal
- Issue #2408[4220] - Stackoverflow detection for linux, e.g. based on libsigsegv
- PR #2377[4221] - Add a customization point for put_parcel so we can override actions
- Issue #2368[4222] - HPX_ASSERT problem
- Issue #2324[4223] - Change default number of threads used to the maximum of the system
- Issue #2266[4224] - hpx_0.9.99 make tests fail
- PR #2195[4225] - Support for code completion in VIM
- Issue #2137[4226] - Hpx does not compile over osx
- Issue #2092[4227] - make tests should just build the tests
- Issue #2026[4228] - Build HPX with Apple's clang

[4206] https://github.com/STEllAR-GROUP/hpx/issues/2566
[4207] https://github.com/STEllAR-GROUP/hpx/issues/2565
[4208] https://github.com/STEllAR-GROUP/hpx/pull/2556
[4209] https://github.com/STEllAR-GROUP/hpx/pull/2546
[4210] https://github.com/STEllAR-GROUP/hpx/issues/2543
[4211] https://github.com/STEllAR-GROUP/hpx/pull/2495
[4212] https://github.com/STEllAR-GROUP/hpx/issues/2480
[4213] https://github.com/STEllAR-GROUP/hpx/issues/2471
[4214] https://github.com/STEllAR-GROUP/hpx/issues/2456
[4215] https://github.com/STEllAR-GROUP/hpx/issues/2455
[4216] https://github.com/STEllAR-GROUP/hpx/pull/2454
[4217] https://github.com/STEllAR-GROUP/hpx/pull/2453
[4218] https://github.com/STEllAR-GROUP/hpx/pull/2447
[4219] https://github.com/STEllAR-GROUP/hpx/issues/2439
[4220] https://github.com/STEllAR-GROUP/hpx/issues/2408
[4221] https://github.com/STEllAR-GROUP/hpx/pull/2377
[4222] https://github.com/STEllAR-GROUP/hpx/issues/2368
[4223] https://github.com/STEllAR-GROUP/hpx/issues/2324
[4224] https://github.com/STEllAR-GROUP/hpx/issues/2266
[4225] https://github.com/STEllAR-GROUP/hpx/pull/2195
[4226] https://github.com/STEllAR-GROUP/hpx/issues/2137
[4227] https://github.com/STEllAR-GROUP/hpx/issues/2092
[4228] https://github.com/STEllAR-GROUP/hpx/issues/2026

- Issue #1932[4229] - hpx with PBS fails on multiple localities
- PR #1914[4230] - Parallel heap algorithm implementations WIP
- Issue #1598[4231] - Disconnecting a locality results in segfault using heartbeat example
- Issue #1404[4232] - unwrapped doesn't work with movable only types
- Issue #1400[4233] - hpx::util::unwrapped doesn't work with non-future types
- Issue #1205[4234] - TSS is broken
- Issue #1126[4235] - vector<future<T> > does not work gracefully with dataflow, when_all and unwrapped
- Issue #1056[4236] - Thread manager cleanup
- Issue #863[4237] - Futures should not require a default constructor
- Issue #856[4238] - Allow runtimemode_connect to be used with security enabled
- Issue #726[4239] - Valgrind
- Issue #701[4240] - Add RCR performance counter component
- Issue #528[4241] - Add support for known failures and warning count/comparisons to hpx_run_tests.py

HPX V1.0.0 (Apr 24, 2017)

General changes

Here are some of the main highlights and changes for this release (in no particular order):

- Added the facility `hpx::split_future` which allows one to convert a `future<tuple<Ts...>>` into a `tuple<future<Ts>...>`. This functionality is not available when compiling *HPX* with VS2012.
- Added a new type of performance counter which allows one to return a list of values for each invocation. We also added a first counter of this type which collects a histogram of the times between parcels being created.
- Added new LCOs: `hpx::lcos::channel` and `hpx::lcos::local::channel` which are very similar to the well known channel constructs used in the Go language.
- Added new performance counters reporting the amount of data handled by the networking layer on a action-by-action basis (please see PR #2289[4242] for more details).
- Added a new facility `hpx::lcos::barrier`, replacing the equally named older one. The new facility has a slightly changed API and is much more efficient. Most notable, the new facility exposes a (global) function `hpx::lcos::barrier::synchronize()` which represents a global barrier across all localities.

[4229] https://github.com/STEllAR-GROUP/hpx/issues/1932
[4230] https://github.com/STEllAR-GROUP/hpx/pull/1914
[4231] https://github.com/STEllAR-GROUP/hpx/issues/1598
[4232] https://github.com/STEllAR-GROUP/hpx/issues/1404
[4233] https://github.com/STEllAR-GROUP/hpx/issues/1400
[4234] https://github.com/STEllAR-GROUP/hpx/issues/1205
[4235] https://github.com/STEllAR-GROUP/hpx/issues/1126
[4236] https://github.com/STEllAR-GROUP/hpx/issues/1056
[4237] https://github.com/STEllAR-GROUP/hpx/issues/863
[4238] https://github.com/STEllAR-GROUP/hpx/issues/856
[4239] https://github.com/STEllAR-GROUP/hpx/issues/726
[4240] https://github.com/STEllAR-GROUP/hpx/issues/701
[4241] https://github.com/STEllAR-GROUP/hpx/issues/528
[4242] https://github.com/STEllAR-GROUP/hpx/pull/2289

- We have started to add support for vectorization to our parallel algorithm implementations. This support depends on using an external library, currently either Vc Library or |boost_simd|_. Please see Issue #2333[4243] for a list of currently supported algorithms. This is an experimental feature and its implementation and/or API might change in the future. Please see this blog-post[4244] for more information.
- The parameter sequence for the `hpx::parallel::transform_reduce` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17. The old overload can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.
- The algorithm `hpx::parallel::inner_product` has been renamed to `hpx::parallel::transform_reduce` to match the changes this algorithm has undergone while being moved to C++17. The old inner_product names can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.
- Added versions of `hpx::get_ptr` taking client side representations for component instances as their parameter (instead of a global id).
- Added the helper utility `hpx::performance_counters::performance_counter_set` helping to encapsulate a set of performance counters to be managed concurrently.
- All execution policies and related classes have been renamed to be consistent with the naming changes applied for C++17. All policies now live in the namespace `hpx::parallel::execution`. The ols names can be still enabled at configure time by specifying `-DHPX_WITH_EXECUTION_POLICY_COMPATIBILITY=On` to CMake.
- The thread scheduling subsystem has undergone a major refactoring which results in significant performance improvements. We have also imroved the performance of creating `hpx::future` and of various facilities handling those.
- We have consolidated all of the code in HPX.Compute related to the integration of CUDA. `hpx::partitioned_vector` has been enabled to be usable with `hpx::compute::vector` which allows one to place the partitions on one or more GPU devices.
- Added new performance counters exposing various internals of the thread scheduling subsystem, such as the current idle- and busy-loop counters and instantaneous scheduler utilization.
- Extended and improved the use of the ITTNotify hooks allowing to collect performance counter data and function annotation information from within the Intel Amplifier tool.

Breaking changes

- We have dropped support for the gcc compiler versions V4.6 and 4.7. The minimal gcc version we now test on is gcc V4.8.
- We have removed (default) support for `boost::chrono` in interfaces, uses of it have been replaced with `std::chrono`. This facility can be still enabled at configure time by specifying `-DHPX_WITH_BOOST_CHRONO_COMPATIBILITY=On` to CMake.
- The parameter sequence for the `hpx::parallel::transform_reduce` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17.
- The algorithm `hpx::parallel::inner_product` has been renamed to `hpx::parallel::transform_reduce` to match the changes this algorithm has undergone while being moved to C++17.

[4243] https://github.com/STEllAR-GROUP/hpx/issues/2333
[4244] http://stellar-group.org/2016/09/vectorized-cpp-parallel-algorithms-with-hpx/

- the build options `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` and `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY` are now disabled by default. Please change your code still depending on the deprecated interfaces.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2596[4245] - Adding apex data
- PR #2595[4246] - Remove obsolete file
- Issue #2594[4247] - FindOpenCL.cmake mismatch with the official cmake module
- PR #2592[4248] - First attempt to introduce spmd_block in hpx
- Issue #2591[4249] - Feature request: continuation (then) which does not require the callable object to take a future<R> as parameter
- PR #2588[4250] - Daint fixes
- PR #2587[4251] - Fixing transfer_(continuation)_action::schedule
- PR #2585[4252] - Work around MSVC having an ICE when compiling with -Ob2
- PR #2583[4253] - changing 7zip command to 7za in roll_release.sh
- PR #2582[4254] - First attempt to introduce spmd_block in hpx
- PR #2581[4255] - Enable annotated function for parallel algorithms
- PR #2580[4256] - First attempt to introduce spmd_block in hpx
- PR #2579[4257] - Make thread NICE level setting an option
- PR #2578[4258] - Implementing enqueue instead of busy wait when no sender is available
- PR #2577[4259] - Retrieve -std=c++11 consistent nvcc flag
- PR #2576[4260] - Add missing dependencies of cuda based tests
- PR #2575[4261] - Remove warnings due to some captured variables
- PR #2573[4262] - Attempt to resolve resolve_locality
- PR #2572[4263] - Adding APEX hooks to background thread

[4245] https://github.com/STEllAR-GROUP/hpx/pull/2596
[4246] https://github.com/STEllAR-GROUP/hpx/pull/2595
[4247] https://github.com/STEllAR-GROUP/hpx/issues/2594
[4248] https://github.com/STEllAR-GROUP/hpx/pull/2592
[4249] https://github.com/STEllAR-GROUP/hpx/issues/2591
[4250] https://github.com/STEllAR-GROUP/hpx/pull/2588
[4251] https://github.com/STEllAR-GROUP/hpx/pull/2587
[4252] https://github.com/STEllAR-GROUP/hpx/pull/2585
[4253] https://github.com/STEllAR-GROUP/hpx/pull/2583
[4254] https://github.com/STEllAR-GROUP/hpx/pull/2582
[4255] https://github.com/STEllAR-GROUP/hpx/pull/2581
[4256] https://github.com/STEllAR-GROUP/hpx/pull/2580
[4257] https://github.com/STEllAR-GROUP/hpx/pull/2579
[4258] https://github.com/STEllAR-GROUP/hpx/pull/2578
[4259] https://github.com/STEllAR-GROUP/hpx/pull/2577
[4260] https://github.com/STEllAR-GROUP/hpx/pull/2576
[4261] https://github.com/STEllAR-GROUP/hpx/pull/2575
[4262] https://github.com/STEllAR-GROUP/hpx/pull/2573
[4263] https://github.com/STEllAR-GROUP/hpx/pull/2572

- PR #2571[4264] - Pick up hpx.ignore_batch_env from config map
- PR #2570[4265] - Add commandline options –hpx:print-counters-locally
- PR #2569[4266] - Fix computeapi unit tests
- PR #2567[4267] - This adds another barrier::synchronize before registering performance counters
- PR #2564[4268] - Cray static toolchain support
- PR #2563[4269] - Fixed unhandled exception during startup
- PR #2562[4270] - Remove partitioned_vector.cu from build tree when nvcc is used
- Issue #2561[4271] - octo-tiger crash with commit 6e921495ff6c26f125d62629cbaad0525f14f7ab
- PR #2560[4272] - Prevent -Wundef warnings on Vc version checks
- PR #2559[4273] - Allowing CUDA callback to set the future directly from an OS thread
- PR #2558[4274] - Remove warnings due to float precisions
- PR #2557[4275] - Removing bogus handling of compile flags for CUDA
- PR #2556[4276] - Fixing scan partitioner
- PR #2554[4277] - Add more diagnostics to error thrown from find_appropriate_destination
- Issue #2555[4278] - No valid parcelport configured
- PR #2553[4279] - Add cmake cuda_arch option
- PR #2552[4280] - Remove incomplete datapar bindings to libflatarray
- PR #2551[4281] - Rename hwloc_topology to hwloc_topology_info
- PR #2550[4282] - Apex api updates
- PR #2549[4283] - Pre-include defines.hpp to get the macro HPX_HAVE_CUDA value
- PR #2548[4284] - Fixing issue with disconnect
- PR #2546[4285] - Some fixes around cuda clang partitioned_vector example
- PR #2545[4286] - Fix uses of the Vc2 datapar flags; the value, not the type, should be passed to functions

[4264] https://github.com/STEllAR-GROUP/hpx/pull/2571
[4265] https://github.com/STEllAR-GROUP/hpx/pull/2570
[4266] https://github.com/STEllAR-GROUP/hpx/pull/2569
[4267] https://github.com/STEllAR-GROUP/hpx/pull/2567
[4268] https://github.com/STEllAR-GROUP/hpx/pull/2564
[4269] https://github.com/STEllAR-GROUP/hpx/pull/2563
[4270] https://github.com/STEllAR-GROUP/hpx/pull/2562
[4271] https://github.com/STEllAR-GROUP/hpx/issues/2561
[4272] https://github.com/STEllAR-GROUP/hpx/pull/2560
[4273] https://github.com/STEllAR-GROUP/hpx/pull/2559
[4274] https://github.com/STEllAR-GROUP/hpx/pull/2558
[4275] https://github.com/STEllAR-GROUP/hpx/pull/2557
[4276] https://github.com/STEllAR-GROUP/hpx/pull/2556
[4277] https://github.com/STEllAR-GROUP/hpx/pull/2554
[4278] https://github.com/STEllAR-GROUP/hpx/issues/2555
[4279] https://github.com/STEllAR-GROUP/hpx/pull/2553
[4280] https://github.com/STEllAR-GROUP/hpx/pull/2552
[4281] https://github.com/STEllAR-GROUP/hpx/pull/2551
[4282] https://github.com/STEllAR-GROUP/hpx/pull/2550
[4283] https://github.com/STEllAR-GROUP/hpx/pull/2549
[4284] https://github.com/STEllAR-GROUP/hpx/pull/2548
[4285] https://github.com/STEllAR-GROUP/hpx/pull/2546
[4286] https://github.com/STEllAR-GROUP/hpx/pull/2545

- PR #2542[4287] - Make HPX_WITH_MALLOC easier to use
- PR #2541[4288] - avoid recompiles when enabling/disabling examples
- PR #2540[4289] - Fixing usage of target_link_libraries()
- PR #2539[4290] - fix RPATH behaviour
- Issue #2538[4291] - HPX_WITH_CUDA corrupts compilation flags
- PR #2537[4292] - Add output of a Bazel Skylark extension for paths and compile options
- PR #2536[4293] - Add counter exposing total available memory to Windows as well
- PR #2535[4294] - Remove obsolete support for security
- Issue #2534[4295] - Remove command line option `--hpx:run-agas-server`
- PR #2533[4296] - Pre-cache locality endpoints during bootstrap
- PR #2532[4297] - Fixing handling of GIDs during serialization preprocessing
- PR #2531[4298] - Amend uses of the term "functor"
- PR #2529[4299] - added counter for reading available memory
- PR #2527[4300] - Facilities to create actions from lambdas
- PR #2526[4301] - Updated docs: HPX_WITH_EXAMPLES
- PR #2525[4302] - Remove warnings related to unused captured variables
- Issue #2524[4303] - CMAKE failed because it is missing: TCMALLOC_LIBRARY TCMALLOC_INCLUDE_DIR
- PR #2523[4304] - Fixing compose_cb stack overflow
- PR #2522[4305] - Instead of unlocking, ignore the lock while creating the message handler
- PR #2521[4306] - Create `LPROGRESS_` logging macro to simplify progress tracking and timings
- PR #2520[4307] - Intel 17 support
- PR #2519[4308] - Fix components example
- PR #2518[4309] - Fixing parcel scheduling

4287 https://github.com/STEllAR-GROUP/hpx/pull/2542
4288 https://github.com/STEllAR-GROUP/hpx/pull/2541
4289 https://github.com/STEllAR-GROUP/hpx/pull/2540
4290 https://github.com/STEllAR-GROUP/hpx/pull/2539
4291 https://github.com/STEllAR-GROUP/hpx/issues/2538
4292 https://github.com/STEllAR-GROUP/hpx/pull/2537
4293 https://github.com/STEllAR-GROUP/hpx/pull/2536
4294 https://github.com/STEllAR-GROUP/hpx/pull/2535
4295 https://github.com/STEllAR-GROUP/hpx/issues/2534
4296 https://github.com/STEllAR-GROUP/hpx/pull/2533
4297 https://github.com/STEllAR-GROUP/hpx/pull/2532
4298 https://github.com/STEllAR-GROUP/hpx/pull/2531
4299 https://github.com/STEllAR-GROUP/hpx/pull/2529
4300 https://github.com/STEllAR-GROUP/hpx/pull/2527
4301 https://github.com/STEllAR-GROUP/hpx/pull/2526
4302 https://github.com/STEllAR-GROUP/hpx/pull/2525
4303 https://github.com/STEllAR-GROUP/hpx/issues/2524
4304 https://github.com/STEllAR-GROUP/hpx/pull/2523
4305 https://github.com/STEllAR-GROUP/hpx/pull/2522
4306 https://github.com/STEllAR-GROUP/hpx/pull/2521
4307 https://github.com/STEllAR-GROUP/hpx/pull/2520
4308 https://github.com/STEllAR-GROUP/hpx/pull/2519
4309 https://github.com/STEllAR-GROUP/hpx/pull/2518

- Issue #2517[4310] - Race condition during Parcel Coalescing Handler creation
- Issue #2516[4311] - HPX locks up when using at least 256 localities
- Issue #2515[4312] - error: Install cannot find "/lib/hpx/libparcel_coalescing.so.0.9.99" but I can see that file
- PR #2514[4313] - Making sure that all continuations of a shared_future are invoked in order
- PR #2513[4314] - Fixing locks held during suspension
- PR #2512[4315] - MPI Parcelport improvements and fixes related to the background work changes
- PR #2511[4316] - Fixing bit-wise (zero-copy) serialization
- Issue #2509[4317] - Linking errors in hwloc_topology
- PR #2508[4318] - Added documentation for debugging with core files
- PR #2506[4319] - Fixing background work invocations
- PR #2505[4320] - Fix tuple serialization
- Issue #2504[4321] - Ensure continuations are called in the order they have been attached
- PR #2503[4322] - Adding serialization support for Vc v2 (datapar)
- PR #2502[4323] - Resolve various, minor compiler warnings
- PR #2501[4324] - Some other fixes around cuda examples
- Issue #2500[4325] - nvcc / cuda clang issue due to a missing -DHPX_WITH_CUDA flag
- PR #2499[4326] - Adding support for std::array to wait_all and friends
- PR #2498[4327] - Execute background work as HPX thread
- PR #2497[4328] - Fixing configuration options for spinlock-deadlock detection
- PR #2496[4329] - Accounting for different compilers in CrayKNL toolchain file
- PR #2494[4330] - Adding component base class which ties a component instance to a given executor
- PR #2493[4331] - Enable controlling amount of pending threads which must be available to allow thread stealing
- PR #2492[4332] - Adding new command line option –hpx:print-counter-reset

[4310] https://github.com/STEllAR-GROUP/hpx/issues/2517
[4311] https://github.com/STEllAR-GROUP/hpx/issues/2516
[4312] https://github.com/STEllAR-GROUP/hpx/issues/2515
[4313] https://github.com/STEllAR-GROUP/hpx/pull/2514
[4314] https://github.com/STEllAR-GROUP/hpx/pull/2513
[4315] https://github.com/STEllAR-GROUP/hpx/pull/2512
[4316] https://github.com/STEllAR-GROUP/hpx/pull/2511
[4317] https://github.com/STEllAR-GROUP/hpx/issues/2509
[4318] https://github.com/STEllAR-GROUP/hpx/pull/2508
[4319] https://github.com/STEllAR-GROUP/hpx/pull/2506
[4320] https://github.com/STEllAR-GROUP/hpx/pull/2505
[4321] https://github.com/STEllAR-GROUP/hpx/issues/2504
[4322] https://github.com/STEllAR-GROUP/hpx/pull/2503
[4323] https://github.com/STEllAR-GROUP/hpx/pull/2502
[4324] https://github.com/STEllAR-GROUP/hpx/pull/2501
[4325] https://github.com/STEllAR-GROUP/hpx/issues/2500
[4326] https://github.com/STEllAR-GROUP/hpx/pull/2499
[4327] https://github.com/STEllAR-GROUP/hpx/pull/2498
[4328] https://github.com/STEllAR-GROUP/hpx/pull/2497
[4329] https://github.com/STEllAR-GROUP/hpx/pull/2496
[4330] https://github.com/STEllAR-GROUP/hpx/pull/2494
[4331] https://github.com/STEllAR-GROUP/hpx/pull/2493
[4332] https://github.com/STEllAR-GROUP/hpx/pull/2492

- PR #2491[4333] - Resolve ambiguities when compiling with APEX
- PR #2490[4334] - Resuming threads waiting on future with higher priority
- Issue #2489[4335] - nvcc issue because -std=c++11 appears twice
- PR #2488[4336] - Adding performance counters exposing the internal idle and busy-loop counters
- PR #2487[4337] - Allowing for plain suspend to reschedule thread right away
- PR #2486[4338] - Only flag HPX code for CUDA if HPX_WITH_CUDA is set
- PR #2485[4339] - Making thread-queue parameters runtime-configurable
- PR #2484[4340] - Added atomic counter for parcel-destinations
- PR #2483[4341] - Added priority-queue lifo scheduler
- PR #2482[4342] - Changing scheduler to steal only if more than a minimal number of tasks are available
- PR #2481[4343] - Extending command line option –hpx:print-counter-destination to support value 'none'
- PR #2479[4344] - Added option to disable signal handler
- PR #2478[4345] - Making sure the sine performance counter module gets loaded only for the corresponding example
- Issue #2477[4346] - Breaking at a throw statement
- PR #2476[4347] - Annotated function
- PR #2475[4348] - Ensure that using %osthread% during logging will not throw for non-hpx threads
- PR #2474[4349] - Remove now superficial non_direct actions from base_lco and friends
- PR #2473[4350] - Refining support for ITTNotify
- PR #2472[4351] - Some fixes around hpx compute
- Issue #2470[4352] - redefinition of boost::detail::spinlock
- Issue #2469[4353] - Dataflow performance issue
- PR #2468[4354] - Perf docs update
- PR #2466[4355] - Guarantee to execute remote direct actions on HPX-thread

4333 https://github.com/STEllAR-GROUP/hpx/pull/2491
4334 https://github.com/STEllAR-GROUP/hpx/pull/2490
4335 https://github.com/STEllAR-GROUP/hpx/issues/2489
4336 https://github.com/STEllAR-GROUP/hpx/pull/2488
4337 https://github.com/STEllAR-GROUP/hpx/pull/2487
4338 https://github.com/STEllAR-GROUP/hpx/pull/2486
4339 https://github.com/STEllAR-GROUP/hpx/pull/2485
4340 https://github.com/STEllAR-GROUP/hpx/pull/2484
4341 https://github.com/STEllAR-GROUP/hpx/pull/2483
4342 https://github.com/STEllAR-GROUP/hpx/pull/2482
4343 https://github.com/STEllAR-GROUP/hpx/pull/2481
4344 https://github.com/STEllAR-GROUP/hpx/pull/2479
4345 https://github.com/STEllAR-GROUP/hpx/pull/2478
4346 https://github.com/STEllAR-GROUP/hpx/issues/2477
4347 https://github.com/STEllAR-GROUP/hpx/pull/2476
4348 https://github.com/STEllAR-GROUP/hpx/pull/2475
4349 https://github.com/STEllAR-GROUP/hpx/pull/2474
4350 https://github.com/STEllAR-GROUP/hpx/pull/2473
4351 https://github.com/STEllAR-GROUP/hpx/pull/2472
4352 https://github.com/STEllAR-GROUP/hpx/issues/2470
4353 https://github.com/STEllAR-GROUP/hpx/issues/2469
4354 https://github.com/STEllAR-GROUP/hpx/pull/2468
4355 https://github.com/STEllAR-GROUP/hpx/pull/2466

- PR #2465[4356] - Improve demo : Async copy and fixed device handling
- PR #2464[4357] - Adding performance counter exposing instantaneous scheduler utilization
- PR #2463[4358] - Downcast to future<void>
- PR #2462[4359] - Fixed usage of ITT-Notify API with Intel Amplifier
- PR #2461[4360] - Cublas demo
- PR #2460[4361] - Fixing thread bindings
- PR #2459[4362] - Make -std=c++11 nvcc flag consistent for in-build and installed versions
- Issue #2457[4363] - Segmentation fault when registering a partitioned vector
- PR #2452[4364] - Properly releasing global barrier for unhandled exceptions
- PR #2451[4365] - Fixing long shutdown times
- PR #2450[4366] - Attempting to fix initialization errors on newer platforms (Boost V1.63)
- PR #2449[4367] - Replace BOOST_COMPILER_FENCE with an HPX version
- PR #2448[4368] - This fixes a possible race in the migration code
- **PR #2445[4369] - Fixing dataflow et.al. for futures or future-ranges wrapped** into ref()
- PR #2444[4370] - Fix segfaults
- PR #2443[4371] - Issue 2442
- Issue #2442[4372] - Mismatch between #if/#endif and namespace scope brackets in this_thread_executers.hpp
- Issue #2441[4373] - undeclared identifier BOOST_COMPILER_FENCE
- PR #2440[4374] - Knl build
- PR #2438[4375] - Datapar backend
- PR #2437[4376] - Adapt algorithm parameter sequence changes from C++17
- PR #2436[4377] - Adapt execution policy name changes from C++17
- Issue #2435[4378] - Trunk broken, undefined reference to hpx::thread::interrupt(hpx::thread::id, bool)

4356 https://github.com/STEllAR-GROUP/hpx/pull/2465
4357 https://github.com/STEllAR-GROUP/hpx/pull/2464
4358 https://github.com/STEllAR-GROUP/hpx/pull/2463
4359 https://github.com/STEllAR-GROUP/hpx/pull/2462
4360 https://github.com/STEllAR-GROUP/hpx/pull/2461
4361 https://github.com/STEllAR-GROUP/hpx/pull/2460
4362 https://github.com/STEllAR-GROUP/hpx/pull/2459
4363 https://github.com/STEllAR-GROUP/hpx/issues/2457
4364 https://github.com/STEllAR-GROUP/hpx/pull/2452
4365 https://github.com/STEllAR-GROUP/hpx/pull/2451
4366 https://github.com/STEllAR-GROUP/hpx/pull/2450
4367 https://github.com/STEllAR-GROUP/hpx/pull/2449
4368 https://github.com/STEllAR-GROUP/hpx/pull/2448
4369 https://github.com/STEllAR-GROUP/hpx/pull/2445
4370 https://github.com/STEllAR-GROUP/hpx/pull/2444
4371 https://github.com/STEllAR-GROUP/hpx/pull/2443
4372 https://github.com/STEllAR-GROUP/hpx/issues/2442
4373 https://github.com/STEllAR-GROUP/hpx/issues/2441
4374 https://github.com/STEllAR-GROUP/hpx/pull/2440
4375 https://github.com/STEllAR-GROUP/hpx/pull/2438
4376 https://github.com/STEllAR-GROUP/hpx/pull/2437
4377 https://github.com/STEllAR-GROUP/hpx/pull/2436
4378 https://github.com/STEllAR-GROUP/hpx/issues/2435

- PR #2434[4379] - More fixes to resource manager
- PR #2433[4380] - Added versions of `hpx::get_ptr` taking client side representations
- PR #2432[4381] - Warning fixes
- PR #2431[4382] - Adding facility representing set of performance counters
- PR #2430[4383] - Fix parallel_executor thread spawning
- PR #2429[4384] - Fix attribute warning for gcc
- Issue #2427[4385] - Seg fault running octo-tiger with latest HPX commit
- Issue #2426[4386] - Bug in 9592f5c0bc29806fce0dbe73f35b6ca7e027edcb causes immediate crash in Octo-tiger
- PR #2425[4387] - Fix nvcc errors due to constexpr specifier
- Issue #2424[4388] - Async action on component present on hpx::find_here is executing synchronously
- PR #2423[4389] - Fix nvcc errors due to constexpr specifier
- PR #2422[4390] - Implementing hpx::this_thread thread data functions
- PR #2421[4391] - Adding benchmark for wait_all
- Issue #2420[4392] - Returning object of a component client from another component action fails
- PR #2419[4393] - Infiniband parcelport
- Issue #2418[4394] - gcc + nvcc fails to compile code that uses partitioned_vector
- PR #2417[4395] - Fixing context switching
- PR #2416[4396] - Adding fixes and workarounds to allow compilation with nvcc/msvc (VS2015up3)
- PR #2415[4397] - Fix errors coming from hpx compute examples
- PR #2414[4398] - Fixing msvc12
- PR #2413[4399] - Enable cuda/nvcc or cuda/clang when using add_hpx_executable()
- PR #2412[4400] - Fix issue in HPX_SetupTarget.cmake when cuda is used
- PR #2411[4401] - This fixes the core compilation issues with MSVC12

4379 https://github.com/STEllAR-GROUP/hpx/pull/2434
4380 https://github.com/STEllAR-GROUP/hpx/pull/2433
4381 https://github.com/STEllAR-GROUP/hpx/pull/2432
4382 https://github.com/STEllAR-GROUP/hpx/pull/2431
4383 https://github.com/STEllAR-GROUP/hpx/pull/2430
4384 https://github.com/STEllAR-GROUP/hpx/pull/2429
4385 https://github.com/STEllAR-GROUP/hpx/issues/2427
4386 https://github.com/STEllAR-GROUP/hpx/issues/2426
4387 https://github.com/STEllAR-GROUP/hpx/pull/2425
4388 https://github.com/STEllAR-GROUP/hpx/issues/2424
4389 https://github.com/STEllAR-GROUP/hpx/pull/2423
4390 https://github.com/STEllAR-GROUP/hpx/pull/2422
4391 https://github.com/STEllAR-GROUP/hpx/pull/2421
4392 https://github.com/STEllAR-GROUP/hpx/issues/2420
4393 https://github.com/STEllAR-GROUP/hpx/pull/2419
4394 https://github.com/STEllAR-GROUP/hpx/issues/2418
4395 https://github.com/STEllAR-GROUP/hpx/pull/2417
4396 https://github.com/STEllAR-GROUP/hpx/pull/2416
4397 https://github.com/STEllAR-GROUP/hpx/pull/2415
4398 https://github.com/STEllAR-GROUP/hpx/pull/2414
4399 https://github.com/STEllAR-GROUP/hpx/pull/2413
4400 https://github.com/STEllAR-GROUP/hpx/pull/2412
4401 https://github.com/STEllAR-GROUP/hpx/pull/2411

- Issue #2410[4402] - `undefined reference to opal_hwloc191_hwloc_.....`
- PR #2409[4403] - Fixing locking for channel and receive_buffer
- PR #2407[4404] - Solving #2402 and #2403
- PR #2406[4405] - Improve guards
- PR #2405[4406] - Enable parallel::for_each for iterators returning proxy types
- PR #2404[4407] - Forward the explicitly given result_type in the hpx invoke
- Issue #2403[4408] - datapar_execution + zip iterator: lambda arguments aren't references
- Issue #2402[4409] - datapar algorithm instantiated with wrong type #2402
- PR #2401[4410] - Added support for imported libraries to HPX_Libraries.cmake
- PR #2400[4411] - Use CMake policy CMP0060
- Issue #2399[4412] - Error trying to push back vector of futures to vector
- PR #2398[4413] - Allow config #defines to be written out to custom config/defines.hpp
- Issue #2397[4414] - CMake generated config defines can cause tedious rebuilds category
- Issue #2396[4415] - BOOST_ROOT paths are not used at link time
- PR #2395[4416] - Fix target_link_libraries() issue when HPX Cuda is enabled
- Issue #2394[4417] - Template compilation error using HPX_WITH_DATAPAR_LIBFLATARRAY
- PR #2393[4418] - Fixing lock registration for recursive mutex
- PR #2392[4419] - Add keywords in target_link_libraries in hpx_setup_target
- PR #2391[4420] - Clang goroutines
- Issue #2390[4421] - Adapt execution policy name changes from C++17
- PR #2389[4422] - Chunk allocator and pool are not used and are obsolete
- PR #2388[4423] - Adding functionalities to datapar needed by octotiger
- PR #2387[4424] - Fixing race condition for early parcels

[4402] https://github.com/STEllAR-GROUP/hpx/issues/2410
[4403] https://github.com/STEllAR-GROUP/hpx/pull/2409
[4404] https://github.com/STEllAR-GROUP/hpx/pull/2407
[4405] https://github.com/STEllAR-GROUP/hpx/pull/2406
[4406] https://github.com/STEllAR-GROUP/hpx/pull/2405
[4407] https://github.com/STEllAR-GROUP/hpx/pull/2404
[4408] https://github.com/STEllAR-GROUP/hpx/issues/2403
[4409] https://github.com/STEllAR-GROUP/hpx/issues/2402
[4410] https://github.com/STEllAR-GROUP/hpx/pull/2401
[4411] https://github.com/STEllAR-GROUP/hpx/pull/2400
[4412] https://github.com/STEllAR-GROUP/hpx/issues/2399
[4413] https://github.com/STEllAR-GROUP/hpx/pull/2398
[4414] https://github.com/STEllAR-GROUP/hpx/issues/2397
[4415] https://github.com/STEllAR-GROUP/hpx/issues/2396
[4416] https://github.com/STEllAR-GROUP/hpx/pull/2395
[4417] https://github.com/STEllAR-GROUP/hpx/issues/2394
[4418] https://github.com/STEllAR-GROUP/hpx/pull/2393
[4419] https://github.com/STEllAR-GROUP/hpx/pull/2392
[4420] https://github.com/STEllAR-GROUP/hpx/pull/2391
[4421] https://github.com/STEllAR-GROUP/hpx/issues/2390
[4422] https://github.com/STEllAR-GROUP/hpx/pull/2389
[4423] https://github.com/STEllAR-GROUP/hpx/pull/2388
[4424] https://github.com/STEllAR-GROUP/hpx/pull/2387

- Issue #2386[4425] - Lock registration broken for recursive_mutex
- PR #2385[4426] - Datapar zip iterator
- PR #2384[4427] - Fixing race condition in for_loop_reduction
- PR #2383[4428] - Continuations
- PR #2382[4429] - add LibFlatArray-based backend for datapar
- PR #2381[4430] - remove unused typedef to get rid of compiler warnings
- PR #2380[4431] - Tau cleanup
- PR #2379[4432] - Can send immediate
- PR #2378[4433] - Renaming copy_helper/copy_n_helper/move_helper/move_n_helper
- Issue #2376[4434] - Boost trunk's spinlock initializer fails to compile
- PR #2375[4435] - Add support for minimal thread local data
- PR #2374[4436] - Adding API functions set_config_entry_callback
- PR #2373[4437] - Add a simple utility for debugging that gives suspended task backtraces
- PR #2372[4438] - Barrier Fixes
- Issue #2370[4439] - Can't wait on a wrapped future
- PR #2369[4440] - Fixing stable_partition
- PR #2367[4441] - Fixing find_prefixes for Windows platforms
- PR #2366[4442] - Testing for experimental/optional only in C++14 mode
- PR #2364[4443] - Adding set_config_entry
- PR #2363[4444] - Fix papi
- PR #2362[4445] - Adding missing macros for new non-direct actions
- PR #2361[4446] - Improve cmake output to help debug compiler incompatibility check
- PR #2360[4447] - Fixing race condition in condition_variable

4425 https://github.com/STEllAR-GROUP/hpx/issues/2386
4426 https://github.com/STEllAR-GROUP/hpx/pull/2385
4427 https://github.com/STEllAR-GROUP/hpx/pull/2384
4428 https://github.com/STEllAR-GROUP/hpx/pull/2383
4429 https://github.com/STEllAR-GROUP/hpx/pull/2382
4430 https://github.com/STEllAR-GROUP/hpx/pull/2381
4431 https://github.com/STEllAR-GROUP/hpx/pull/2380
4432 https://github.com/STEllAR-GROUP/hpx/pull/2379
4433 https://github.com/STEllAR-GROUP/hpx/pull/2378
4434 https://github.com/STEllAR-GROUP/hpx/issues/2376
4435 https://github.com/STEllAR-GROUP/hpx/pull/2375
4436 https://github.com/STEllAR-GROUP/hpx/pull/2374
4437 https://github.com/STEllAR-GROUP/hpx/pull/2373
4438 https://github.com/STEllAR-GROUP/hpx/pull/2372
4439 https://github.com/STEllAR-GROUP/hpx/issues/2370
4440 https://github.com/STEllAR-GROUP/hpx/pull/2369
4441 https://github.com/STEllAR-GROUP/hpx/pull/2367
4442 https://github.com/STEllAR-GROUP/hpx/pull/2366
4443 https://github.com/STEllAR-GROUP/hpx/pull/2364
4444 https://github.com/STEllAR-GROUP/hpx/pull/2363
4445 https://github.com/STEllAR-GROUP/hpx/pull/2362
4446 https://github.com/STEllAR-GROUP/hpx/pull/2361
4447 https://github.com/STEllAR-GROUP/hpx/pull/2360

- PR #2359[4448] - Fixing shutdown when parcels are still in flight
- Issue #2357[4449] - failed to insert console_print_action into typename_to_id_t registry
- PR #2356[4450] - Fixing return type of get_iterator_tuple
- PR #2355[4451] - Fixing compilation against Boost 1 62
- PR #2354[4452] - Adding serialization for mask_type if CPU_COUNT > 64
- PR #2353[4453] - Adding hooks to tie in APEX into the parcel layer
- Issue #2352[4454] - Compile errors when using intel 17 beta (for KNL) on edison
- PR #2351[4455] - Fix function vtable get_function_address implementation
- Issue #2350[4456] - Build failure - master branch (4de09f5) with Intel Compiler v17
- PR #2349[4457] - Enabling zero-copy serialization support for std::vector<>
- PR #2348[4458] - Adding test to verify #2334 is fixed
- PR #2347[4459] - Bug fixes for hpx.compute and hpx::lcos::channel
- PR #2346[4460] - Removing cmake "find" files that are in the APEX cmake Modules
- PR #2345[4461] - Implemented parallel::stable_partition
- PR #2344[4462] - Making hpx::lcos::channel usable with basename registration
- PR #2343[4463] - Fix a couple of examples that failed to compile after recent api changes
- Issue #2342[4464] - Enabling APEX causes link errors
- PR #2341[4465] - Removing cmake "find" files that are in the APEX cmake Modules
- PR #2340[4466] - Implemented all existing datapar algorithms using Boost.SIMD
- PR #2339[4467] - Fixing 2338
- PR #2338[4468] - Possible race in sliding semaphore
- PR #2337[4469] - Adjust osu_latency test to measure window_size parcels in flight at once
- PR #2336[4470] - Allowing remote direct actions to be executed without spawning a task

[4448] https://github.com/STEllAR-GROUP/hpx/pull/2359
[4449] https://github.com/STEllAR-GROUP/hpx/issues/2357
[4450] https://github.com/STEllAR-GROUP/hpx/pull/2356
[4451] https://github.com/STEllAR-GROUP/hpx/pull/2355
[4452] https://github.com/STEllAR-GROUP/hpx/pull/2354
[4453] https://github.com/STEllAR-GROUP/hpx/pull/2353
[4454] https://github.com/STEllAR-GROUP/hpx/issues/2352
[4455] https://github.com/STEllAR-GROUP/hpx/pull/2351
[4456] https://github.com/STEllAR-GROUP/hpx/issues/2350
[4457] https://github.com/STEllAR-GROUP/hpx/pull/2349
[4458] https://github.com/STEllAR-GROUP/hpx/pull/2348
[4459] https://github.com/STEllAR-GROUP/hpx/pull/2347
[4460] https://github.com/STEllAR-GROUP/hpx/pull/2346
[4461] https://github.com/STEllAR-GROUP/hpx/pull/2345
[4462] https://github.com/STEllAR-GROUP/hpx/pull/2344
[4463] https://github.com/STEllAR-GROUP/hpx/pull/2343
[4464] https://github.com/STEllAR-GROUP/hpx/issues/2342
[4465] https://github.com/STEllAR-GROUP/hpx/pull/2341
[4466] https://github.com/STEllAR-GROUP/hpx/pull/2340
[4467] https://github.com/STEllAR-GROUP/hpx/pull/2339
[4468] https://github.com/STEllAR-GROUP/hpx/pull/2338
[4469] https://github.com/STEllAR-GROUP/hpx/pull/2337
[4470] https://github.com/STEllAR-GROUP/hpx/pull/2336

- PR #2335[4471] - Making sure multiple components are properly initialized from arguments
- Issue #2334[4472] - Cannot construct component with large vector on a remote locality
- PR #2332[4473] - Fixing hpx::lcos::local::barrier
- PR #2331[4474] - Updating APEX support to include OTF2
- PR #2330[4475] - Support for data-parallelism for parallel algorithms
- Issue #2329[4476] - Coordinate settings in cmake
- PR #2328[4477] - fix LibGeoDecomp builds with HPX + GCC 5.3.0 + CUDA 8RC
- PR #2326[4478] - Making scan_partitioner work (for now)
- Issue #2323[4479] - Constructing a vector of components only correctly initializes the first component
- PR #2322[4480] - Fix problems that bubbled up after merging #2278
- PR #2321[4481] - Scalable barrier
- PR #2320[4482] - Std flag fixes
- Issue #2319[4483] - -std=c++14 and -std=c++1y with Intel can't build recent Boost builds due to insufficient C++14 support; don't enable these flags by default for Intel
- PR #2318[4484] - Improve handling of –hpx:bind=<bind-spec>
- PR #2317[4485] - Making sure command line warnings are printed once only
- PR #2316[4486] - Fixing command line handling for default bind mode
- PR #2315[4487] - Set id_retrieved if set_id is present
- Issue #2314[4488] - Warning for requested/allocated thread discrepancy is printed twice
- Issue #2313[4489] - –hpx:print-bind doesn't work with –hpx:pu-step
- Issue #2312[4490] - –hpx:bind range specifier restrictions are overly restrictive
- Issue #2311[4491] - hpx_0.9.99 out of project build fails
- PR #2310[4492] - Simplify function registration
- PR #2309[4493] - Spelling and grammar revisions in documentation (and some code)

[4471] https://github.com/STEllAR-GROUP/hpx/pull/2335
[4472] https://github.com/STEllAR-GROUP/hpx/issues/2334
[4473] https://github.com/STEllAR-GROUP/hpx/pull/2332
[4474] https://github.com/STEllAR-GROUP/hpx/pull/2331
[4475] https://github.com/STEllAR-GROUP/hpx/pull/2330
[4476] https://github.com/STEllAR-GROUP/hpx/issues/2329
[4477] https://github.com/STEllAR-GROUP/hpx/pull/2328
[4478] https://github.com/STEllAR-GROUP/hpx/pull/2326
[4479] https://github.com/STEllAR-GROUP/hpx/issues/2323
[4480] https://github.com/STEllAR-GROUP/hpx/pull/2322
[4481] https://github.com/STEllAR-GROUP/hpx/pull/2321
[4482] https://github.com/STEllAR-GROUP/hpx/pull/2320
[4483] https://github.com/STEllAR-GROUP/hpx/issues/2319
[4484] https://github.com/STEllAR-GROUP/hpx/pull/2318
[4485] https://github.com/STEllAR-GROUP/hpx/pull/2317
[4486] https://github.com/STEllAR-GROUP/hpx/pull/2316
[4487] https://github.com/STEllAR-GROUP/hpx/pull/2315
[4488] https://github.com/STEllAR-GROUP/hpx/issues/2314
[4489] https://github.com/STEllAR-GROUP/hpx/issues/2313
[4490] https://github.com/STEllAR-GROUP/hpx/issues/2312
[4491] https://github.com/STEllAR-GROUP/hpx/issues/2311
[4492] https://github.com/STEllAR-GROUP/hpx/pull/2310
[4493] https://github.com/STEllAR-GROUP/hpx/pull/2309

- PR #2306[4494] - Correct minor typo in the documentation
- PR #2305[4495] - Cleaning up and fixing parcel coalescing
- PR #2304[4496] - Inspect checks for stream related includes
- PR #2303[4497] - Add functionality allowing to enumerate threads of given state
- PR #2301[4498] - Algorithm overloads fix for VS2013
- PR #2300[4499] - Use <cstdint>, add inspect checks
- PR #2299[4500] - Replace boost::[c]ref with std::[c]ref, add inspect checks
- PR #2297[4501] - Fixing compilation with no hw_loc
- PR #2296[4502] - Hpx compute
- PR #2295[4503] - Making sure for_loop(execution::par, 0, N, ...) is actually executed in parallel
- PR #2294[4504] - Throwing exceptions if the runtime is not up and running
- PR #2293[4505] - Removing unused parcel port code
- PR #2292[4506] - Refactor function vtables
- PR #2291[4507] - Fixing 2286
- PR #2290[4508] - Simplify algorithm overloads
- PR #2289[4509] - Adding performance counters reporting parcel related data on a per-action basis
- Issue #2288[4510] - Remove dormant parcelports
- Issue #2286[4511] - adjustments to parcel handling to support parcelports that do not need a connection cache
- PR #2285[4512] - add CMake option to disable package export
- PR #2283[4513] - Add more inspect checks for use of deprecated components
- Issue #2282[4514] - Arithmetic exception in executor static chunker
- Issue #2281[4515] - For loop doesn't parallelize
- PR #2280[4516] - Fixing 2277: build failure with PAPI

4494 https://github.com/STEllAR-GROUP/hpx/pull/2306
4495 https://github.com/STEllAR-GROUP/hpx/pull/2305
4496 https://github.com/STEllAR-GROUP/hpx/pull/2304
4497 https://github.com/STEllAR-GROUP/hpx/pull/2303
4498 https://github.com/STEllAR-GROUP/hpx/pull/2301
4499 https://github.com/STEllAR-GROUP/hpx/pull/2300
4500 https://github.com/STEllAR-GROUP/hpx/pull/2299
4501 https://github.com/STEllAR-GROUP/hpx/pull/2297
4502 https://github.com/STEllAR-GROUP/hpx/pull/2296
4503 https://github.com/STEllAR-GROUP/hpx/pull/2295
4504 https://github.com/STEllAR-GROUP/hpx/pull/2294
4505 https://github.com/STEllAR-GROUP/hpx/pull/2293
4506 https://github.com/STEllAR-GROUP/hpx/pull/2292
4507 https://github.com/STEllAR-GROUP/hpx/pull/2291
4508 https://github.com/STEllAR-GROUP/hpx/pull/2290
4509 https://github.com/STEllAR-GROUP/hpx/pull/2289
4510 https://github.com/STEllAR-GROUP/hpx/issues/2288
4511 https://github.com/STEllAR-GROUP/hpx/issues/2286
4512 https://github.com/STEllAR-GROUP/hpx/pull/2285
4513 https://github.com/STEllAR-GROUP/hpx/pull/2283
4514 https://github.com/STEllAR-GROUP/hpx/issues/2282
4515 https://github.com/STEllAR-GROUP/hpx/issues/2281
4516 https://github.com/STEllAR-GROUP/hpx/pull/2280

- PR #2279[4517] - Child vs parent stealing
- Issue #2277[4518] - master branch build failure (53c5b4f) with papi
- PR #2276[4519] - Compile time launch policies
- PR #2275[4520] - Replace boost::chrono with std::chrono in interfaces
- PR #2274[4521] - Replace most uses of Boost.Assign with initializer list
- PR #2273[4522] - Fixed typos
- PR #2272[4523] - Inspect checks
- PR #2270[4524] - Adding test verifying -Ihpx.os_threads=all
- PR #2269[4525] - Added inspect check for now obsolete boost type traits
- PR #2268[4526] - Moving more code into source files
- Issue #2267[4527] - Add inspect support to deprecate Boost.TypeTraits
- PR #2265[4528] - Adding channel LCO
- PR #2264[4529] - Make support for std::ref mandatory
- PR #2263[4530] - Constrain tuple_member forwarding constructor
- Issue #2262[4531] - Test hpx.os_threads=all
- Issue #2261[4532] - OS X: Error: no matching constructor for initialization of 'hpx::lcos::local::condition_variable_any'
- Issue #2260[4533] - Make support for std::ref mandatory
- PR #2259[4534] - Remove most of Boost.MPL, Boost.EnableIf and Boost.TypeTraits
- PR #2258[4535] - Fixing #2256
- PR #2257[4536] - Fixing launch process
- Issue #2256[4537] - Actions are not registered if not invoked
- PR #2255[4538] - Coalescing histogram
- PR #2254[4539] - Silence explicit initialization in copy-constructor warnings

4517 https://github.com/STEllAR-GROUP/hpx/pull/2279
4518 https://github.com/STEllAR-GROUP/hpx/issues/2277
4519 https://github.com/STEllAR-GROUP/hpx/pull/2276
4520 https://github.com/STEllAR-GROUP/hpx/pull/2275
4521 https://github.com/STEllAR-GROUP/hpx/pull/2274
4522 https://github.com/STEllAR-GROUP/hpx/pull/2273
4523 https://github.com/STEllAR-GROUP/hpx/pull/2272
4524 https://github.com/STEllAR-GROUP/hpx/pull/2270
4525 https://github.com/STEllAR-GROUP/hpx/pull/2269
4526 https://github.com/STEllAR-GROUP/hpx/pull/2268
4527 https://github.com/STEllAR-GROUP/hpx/issues/2267
4528 https://github.com/STEllAR-GROUP/hpx/pull/2265
4529 https://github.com/STEllAR-GROUP/hpx/pull/2264
4530 https://github.com/STEllAR-GROUP/hpx/pull/2263
4531 https://github.com/STEllAR-GROUP/hpx/issues/2262
4532 https://github.com/STEllAR-GROUP/hpx/issues/2261
4533 https://github.com/STEllAR-GROUP/hpx/issues/2260
4534 https://github.com/STEllAR-GROUP/hpx/pull/2259
4535 https://github.com/STEllAR-GROUP/hpx/pull/2258
4536 https://github.com/STEllAR-GROUP/hpx/pull/2257
4537 https://github.com/STEllAR-GROUP/hpx/issues/2256
4538 https://github.com/STEllAR-GROUP/hpx/pull/2255
4539 https://github.com/STEllAR-GROUP/hpx/pull/2254

- PR #2253[4540] - Drop support for GCC 4.6 and 4.7
- PR #2252[4541] - Prepare V1.0
- PR #2251[4542] - Convert to 0.9.99
- PR #2249[4543] - Adding iterator_facade and iterator_adaptor
- Issue #2248[4544] - Need a feature to yield to a new task immediately
- PR #2246[4545] - Adding split_future
- PR #2245[4546] - Add an example for handing over a component instance to a dynamically launched locality
- Issue #2243[4547] - Add example demonstrating AGAS symbolic name registration
- Issue #2242[4548] - pkgconfig test broken on CentOS 7 / Boost 1.61
- Issue #2241[4549] - Compilation error for partitioned vector in hpx_compute branch
- PR #2240[4550] - Fixing termination detection on one locality
- Issue #2239[4551] - Create a new facility lcos::split_all
- Issue #2236[4552] - hpx::cout vs. std::cout
- PR #2232[4553] - Implement local-only primary namespace service
- Issue #2147[4554] - would like to know how much data is being routed by particular actions
- Issue #2109[4555] - Warning while compiling hpx
- Issue #1973[4556] - Setting INTERFACE_COMPILE_OPTIONS for hpx_init in CMake taints Fortran_FLAGS
- Issue #1864[4557] - run_guarded using bound function ignores reference
- Issue #1754[4558] - Running with TCP parcelport causes immediate crash or freeze
- Issue #1655[4559] - Enable zip_iterator to be used with Boost traversal iterator categories
- Issue #1591[4560] - Optimize AGAS for shared memory only operation
- Issue #1401[4561] - Need an efficient infiniband parcelport
- Issue #1125[4562] - Fix the IPC parcelport

4540 https://github.com/STEllAR-GROUP/hpx/pull/2253
4541 https://github.com/STEllAR-GROUP/hpx/pull/2252
4542 https://github.com/STEllAR-GROUP/hpx/pull/2251
4543 https://github.com/STEllAR-GROUP/hpx/pull/2249
4544 https://github.com/STEllAR-GROUP/hpx/issues/2248
4545 https://github.com/STEllAR-GROUP/hpx/pull/2246
4546 https://github.com/STEllAR-GROUP/hpx/pull/2245
4547 https://github.com/STEllAR-GROUP/hpx/issues/2243
4548 https://github.com/STEllAR-GROUP/hpx/issues/2242
4549 https://github.com/STEllAR-GROUP/hpx/issues/2241
4550 https://github.com/STEllAR-GROUP/hpx/pull/2240
4551 https://github.com/STEllAR-GROUP/hpx/issues/2239
4552 https://github.com/STEllAR-GROUP/hpx/issues/2236
4553 https://github.com/STEllAR-GROUP/hpx/pull/2232
4554 https://github.com/STEllAR-GROUP/hpx/issues/2147
4555 https://github.com/STEllAR-GROUP/hpx/issues/2109
4556 https://github.com/STEllAR-GROUP/hpx/issues/1973
4557 https://github.com/STEllAR-GROUP/hpx/issues/1864
4558 https://github.com/STEllAR-GROUP/hpx/issues/1754
4559 https://github.com/STEllAR-GROUP/hpx/issues/1655
4560 https://github.com/STEllAR-GROUP/hpx/issues/1591
4561 https://github.com/STEllAR-GROUP/hpx/issues/1401
4562 https://github.com/STEllAR-GROUP/hpx/issues/1125

- Issue #839[4563] - Refactor ibverbs and shmem parcelport
- Issue #702[4564] - Add instrumentation of parcel layer
- Issue #668[4565] - Implement ispc task interface
- Issue #533[4566] - Thread queue/deque internal parameters should be runtime configurable
- Issue #475[4567] - Create a means of combining performance counters into querysets

HPX V0.9.99 (Jul 15, 2016)

General changes

As the version number of this release hints, we consider this release to be a preview for the upcoming *HPX* V1.0. All of the functionalities we set out to implement for V1.0 are in place; all of the features we wanted to have exposed are ready. We are very happy with the stability and performance of *HPX* and we would like to present this release to the community in order for us to gather broad feedback before releasing V1.0. We still expect for some minor details to change, but on the whole this release represents what we would like to have in a V1.0.

Overall, since the last release we have had almost 1600 commits while closing almost 400 tickets. These numbers reflect the incredible development activity we have seen over the last couple of months. We would like to express a big 'Thank you!' to all contributors and those who helped to make this release happen.

The most notable addition in terms of new functionality available with this release is the full implementation of object migration (i.e. the ability to transparently move *HPX* components to a different compute node). Additionally, this release of *HPX* cleans up many minor issues and some API inconsistencies.

Here are some of the main highlights and changes for this release (in no particular order):

- We have fixed a couple of issues in AGAS and the parcel layer which have caused hangs, segmentation faults at exit, and a slowdown of applications over time. Fixing those has significantly increased the overall stability and performance of distributed runs.
- We have started to add parallel algorithm overloads based on the C++ Extensions for Ranges (N4560[4568]) proposal. This also includes the addition of projections to the existing algorithms. Please see Issue #1668[4569] for a list of algorithms which have been adapted to N4560[4570].
- We have implemented index-based parallel for-loops based on a corresponding standardization proposal (P0075R1[4571]). Please see Issue #2016[4572] for a list of available algorithms.
- We have added implementations for more parallel algorithms as proposed for the upcoming C++ 17 Standard. See Issue #1141[4573] for an overview of which algorithms are available by now.
- We have started to implement a new prototypical functionality with *HPX.Compute* which uniformly exposes some of the higher level APIs to heterogeneous architectures (currently CUDA). This functionality is an early preview and should not be considered stable. It may change considerably in the future.
- We have pervasively added (optional) executor arguments to all API functions which schedule new work. Executors are now used throughout the code base as the main means of executing tasks.

[4563] https://github.com/STEllAR-GROUP/hpx/issues/839
[4564] https://github.com/STEllAR-GROUP/hpx/issues/702
[4565] https://github.com/STEllAR-GROUP/hpx/issues/668
[4566] https://github.com/STEllAR-GROUP/hpx/issues/533
[4567] https://github.com/STEllAR-GROUP/hpx/issues/475
[4568] http://wg21.link/n4560
[4569] https://github.com/STEllAR-GROUP/hpx/issues/1668
[4570] http://wg21.link/n4560
[4571] http://wg21.link/p0075r1
[4572] https://github.com/STEllAR-GROUP/hpx/issues/2016
[4573] https://github.com/STEllAR-GROUP/hpx/issues/1141

- Added `hpx::make_future<R>(future<T> &&)` allowing to convert a future of any type `T` into a future of any other type `R`, either based on default conversion rules of the embedded types or using a given explicit conversion function.
- We finally finished the implementation of transparent migration of components to another locality. It is now possible to trigger a migration operation without 'stopping the world' for the object to migrate. *HPX* will make sure that no work is being performed on an object before it is migrated and that all subsequently scheduled work for the migrated object will be transparently forwarded to the new locality. Please note that the global id of the migrated object does not change, thus the application will not have to be changed in any way to support this new functionality. Please note that this feature is currently considered experimental. See Issue #559[4574] and PR #1966[4575] for more details.
- The `hpx::dataflow` facility is now usable with actions. Similarly to `hpx::async`, actions can be specified as an explicit template argument (`hpx::dataflow<Action>(target, ...)`) or as the first argument (`hpx::dataflow(Action(), target, ...)`). We have also enabled the use of distribution policies as the target for dataflow invocations. Please see Issue #1265[4576] and PR #1912[4577] for more information.
- Adding overloads of `gather_here` and `gather_there` to accept the plain values of the data to gather (in addition to the existing overloads expecting futures).
- We have cleaned up and refactored large parts of the code base. This helped reducing compile and link times of *HPX* itself and also of applications depending on it. We have further decreased the dependency of *HPX* on the Boost libraries by replacing part of those with facilities available from the standard libraries.
- Wherever possible we have removed dependencies of our API on Boost by replacing those with the equivalent facility from the C++11 standard library.
- We have added new performance counters for parcel coalescing, file-IO, the AGAS cache, and overall scheduler time. Resetting performance counters has been overhauled and fixed.
- We have introduced a generic client type `hpx::components::client<>` and added support for using it with `hpx::async`. This removes the necessity to implement specific client types for every component type without losing type safety. This deemphasizes the need for using the low level `hpx::id_type` for referencing (possibly remote) component instances. The plan is to deprecate the direct use of `hpx::id_type` in user code in the future.
- We have added a special iterator which supports automatic prefetching of one or more arrays for speeding up loop-like code (see `hpx::parallel::util::make_prefetcher_context()`).
- We have extended the interfaces exposed from executors (as proposed by N4406[4578]) to accept an arbitrary number of arguments.

Breaking changes

- In order to move the dataflow facility to `namespace hpx` we added a definition of `hpx::dataflow` which might create ambiguities in existing codes. The previous definition of this facility (`hpx::lcos::local::dataflow`) has been deprecated and is available only if the constant `-DHPX_WITH_LOCAL_DATAFLOW_COMPATIBILITY=On` to CMake[4579] is defined at configuration time. Please explicitly qualify all uses of the dataflow facility if you enable this compatibility setting and encounter ambiguities.
- The adaptation of the C++ Extensions for Ranges (N4560[4580]) proposal imposes some breaking changes related

4574 https://github.com/STEllAR-GROUP/hpx/issues/559
4575 https://github.com/STEllAR-GROUP/hpx/pull/1966
4576 https://github.com/STEllAR-GROUP/hpx/issues/1265
4577 https://github.com/STEllAR-GROUP/hpx/pull/1912
4578 http://wg21.link/n4406
4579 https://www.cmake.org
4580 http://wg21.link/n4560

to the return types of some of the parallel algorithms. Please see Issue #1668[4581] for a list of algorithms which have already been adapted.

- The facility `hpx::lcos::make_future_void()` has been replaced by `hpx::make_future<void>()`.
- We have removed support for Intel V13 and gcc 4.4.x.
- We have removed (default) support for the generic `hpx::parallel::execution_poliy` because it was removed from the Parallelism TS (__cpp11_n4104__) while it was being added to the upcoming C++17 Standard. This facility can be still enabled at configure time by specifying `-DHPX_WITH_GENERIC_EXECUTION_POLICY=On` to CMake.
- Uses of `boost::shared_ptr` and related facilities have been replaced with `std::shared_ptr` and friends. Uses of `boost::unique_lock`, `boost::lock_guard` etc. have also been replaced by the equivalent (and equally named) tools available from the C++11 standard library.
- Facilities that used to expect an explicit `boost::unique_lock` now take an `std::unique_lock`. Additionally, `condition_variable` no longer aliases `condition_variable_any`; its interface now only works with `std::unique_lock<local::mutex>`.
- Uses of `boost::function`, `boost::bind`, `boost::tuple` have been replaced by the corresponding facilities in *HPX* (`hpx::util::function`, `hpx::util::bind`, and `hpx::util::tuple`, respectively).

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2250[4582] - change default chunker of parallel executor to static one
- PR #2247[4583] - HPX on ppc64le
- PR #2244[4584] - Fixing MSVC problems
- PR #2238[4585] - Fixing small typos
- PR #2237[4586] - Fixing small typos
- PR #2234[4587] - Fix broken add test macro when extra args are passed in
- PR #2231[4588] - Fixing possible race during future awaiting in serialization
- PR #2230[4589] - Fix stream nvcc
- PR #2229[4590] - Fixed run_as_hpx_thread
- PR #2228[4591] - On prefetching_test branch : adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2227[4592] - Support for HPXCL's opencl::event
- PR #2226[4593] - Preparing for release of V0.9.99

[4581] https://github.com/STEllAR-GROUP/hpx/issues/1668
[4582] https://github.com/STEllAR-GROUP/hpx/pull/2250
[4583] https://github.com/STEllAR-GROUP/hpx/pull/2247
[4584] https://github.com/STEllAR-GROUP/hpx/pull/2244
[4585] https://github.com/STEllAR-GROUP/hpx/pull/2238
[4586] https://github.com/STEllAR-GROUP/hpx/pull/2237
[4587] https://github.com/STEllAR-GROUP/hpx/pull/2234
[4588] https://github.com/STEllAR-GROUP/hpx/pull/2231
[4589] https://github.com/STEllAR-GROUP/hpx/pull/2230
[4590] https://github.com/STEllAR-GROUP/hpx/pull/2229
[4591] https://github.com/STEllAR-GROUP/hpx/pull/2228
[4592] https://github.com/STEllAR-GROUP/hpx/pull/2227
[4593] https://github.com/STEllAR-GROUP/hpx/pull/2226

- PR #2225[4594] - fix issue when compiling components with hpxcxx
- PR #2224[4595] - Compute alloc fix
- PR #2223[4596] - Simplify promise
- PR #2222[4597] - Replace last uses of boost::function by util::function_nonser
- PR #2221[4598] - Fix config tests
- PR #2220[4599] - Fixing gcc 4.6 compilation issues
- PR #2219[4600] - nullptr support for `[unique_]function`
- PR #2218[4601] - Introducing clang tidy
- PR #2216[4602] - Replace NULL with nullptr
- Issue #2214[4603] - Let inspect flag use of NULL, suggest nullptr instead
- PR #2213[4604] - Require support for nullptr
- PR #2212[4605] - Properly find jemalloc through pkg-config
- PR #2211[4606] - Disable a couple of warnings reported by Intel on Windows
- PR #2210[4607] - Fixed host::block_allocator::bulk_construct
- PR #2209[4608] - Started to clean up new sort algorithms, made things compile for sort_by_key
- PR #2208[4609] - A couple of fixes that were exposed by a new sort algorithm
- PR #2207[4610] - Adding missing includes in /hpx/include/serialization.hpp
- PR #2206[4611] - Call package_action::get_future before package_action::apply
- PR #2205[4612] - The indirect_packaged_task::operator() needs to be run on a HPX thread
- PR #2204[4613] - Variadic executor parameters
- PR #2203[4614] - Delay-initialize members of partitioned iterator
- PR #2202[4615] - Added segmented fill for hpx::vector
- Issue #2201[4616] - Null Thread id encountered on partitioned_vector

[4594] https://github.com/STEllAR-GROUP/hpx/pull/2225
[4595] https://github.com/STEllAR-GROUP/hpx/pull/2224
[4596] https://github.com/STEllAR-GROUP/hpx/pull/2223
[4597] https://github.com/STEllAR-GROUP/hpx/pull/2222
[4598] https://github.com/STEllAR-GROUP/hpx/pull/2221
[4599] https://github.com/STEllAR-GROUP/hpx/pull/2220
[4600] https://github.com/STEllAR-GROUP/hpx/pull/2219
[4601] https://github.com/STEllAR-GROUP/hpx/pull/2218
[4602] https://github.com/STEllAR-GROUP/hpx/pull/2216
[4603] https://github.com/STEllAR-GROUP/hpx/issues/2214
[4604] https://github.com/STEllAR-GROUP/hpx/pull/2213
[4605] https://github.com/STEllAR-GROUP/hpx/pull/2212
[4606] https://github.com/STEllAR-GROUP/hpx/pull/2211
[4607] https://github.com/STEllAR-GROUP/hpx/pull/2210
[4608] https://github.com/STEllAR-GROUP/hpx/pull/2209
[4609] https://github.com/STEllAR-GROUP/hpx/pull/2208
[4610] https://github.com/STEllAR-GROUP/hpx/pull/2207
[4611] https://github.com/STEllAR-GROUP/hpx/pull/2206
[4612] https://github.com/STEllAR-GROUP/hpx/pull/2205
[4613] https://github.com/STEllAR-GROUP/hpx/pull/2204
[4614] https://github.com/STEllAR-GROUP/hpx/pull/2203
[4615] https://github.com/STEllAR-GROUP/hpx/pull/2202
[4616] https://github.com/STEllAR-GROUP/hpx/issues/2201

- PR #2200[4617] - Fix hangs
- PR #2199[4618] - Deprecating hpx/traits.hpp
- PR #2198[4619] - Making explicit inclusion of external libraries into build
- PR #2197[4620] - Fix typo in QT CMakeLists
- PR #2196[4621] - Fixing a gcc warning about attributes being ignored
- PR #2194[4622] - Fixing partitioned_vector_spmd_foreach example
- Issue #2193[4623] - partitioned_vector_spmd_foreach seg faults
- PR #2192[4624] - Support Boost.Thread v4
- PR #2191[4625] - HPX.Compute prototype
- PR #2190[4626] - Spawning operation on new thread if remaining stack space becomes too small
- PR #2189[4627] - Adding callback taking index and future to when_each
- PR #2188[4628] - Adding new example demonstrating receive_buffer
- PR #2187[4629] - Mask 128-bit ints if CUDA is being used
- PR #2186[4630] - Make startup & shutdown functions unique_function
- PR #2185[4631] - Fixing logging output not to cause hang on shutdown
- PR #2184[4632] - Allowing component clients as action return types
- Issue #2183[4633] - Enabling logging output causes hang on shutdown
- Issue #2182[4634] - 1d_stencil seg fault
- Issue #2181[4635] - Setting small stack size does not change default
- PR #2180[4636] - Changing default bind mode to balanced
- PR #2179[4637] - adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2177[4638] - Fixing 2176
- Issue #2176[4639] - Launch process test fails on OSX

4617 https://github.com/STEllAR-GROUP/hpx/pull/2200
4618 https://github.com/STEllAR-GROUP/hpx/pull/2199
4619 https://github.com/STEllAR-GROUP/hpx/pull/2198
4620 https://github.com/STEllAR-GROUP/hpx/pull/2197
4621 https://github.com/STEllAR-GROUP/hpx/pull/2196
4622 https://github.com/STEllAR-GROUP/hpx/pull/2194
4623 https://github.com/STEllAR-GROUP/hpx/issues/2193
4624 https://github.com/STEllAR-GROUP/hpx/pull/2192
4625 https://github.com/STEllAR-GROUP/hpx/pull/2191
4626 https://github.com/STEllAR-GROUP/hpx/pull/2190
4627 https://github.com/STEllAR-GROUP/hpx/pull/2189
4628 https://github.com/STEllAR-GROUP/hpx/pull/2188
4629 https://github.com/STEllAR-GROUP/hpx/pull/2187
4630 https://github.com/STEllAR-GROUP/hpx/pull/2186
4631 https://github.com/STEllAR-GROUP/hpx/pull/2185
4632 https://github.com/STEllAR-GROUP/hpx/pull/2184
4633 https://github.com/STEllAR-GROUP/hpx/issues/2183
4634 https://github.com/STEllAR-GROUP/hpx/issues/2182
4635 https://github.com/STEllAR-GROUP/hpx/issues/2181
4636 https://github.com/STEllAR-GROUP/hpx/pull/2180
4637 https://github.com/STEllAR-GROUP/hpx/pull/2179
4638 https://github.com/STEllAR-GROUP/hpx/pull/2177
4639 https://github.com/STEllAR-GROUP/hpx/issues/2176

- PR #2175[4640] - Fix unbalanced config/warnings includes, add some new ones
- PR #2174[4641] - Fix test categorization : regression not unit
- Issue #2172[4642] - Different performance results
- Issue #2171[4643] - "negative entry in reference count table" running octotiger on 32 nodes on queenbee
- Issue #2170[4644] - Error while compiling on Mac + boost 1.60
- PR #2168[4645] - Fixing problems with is_bitwise_serializable
- Issue #2167[4646] - startup & shutdown function should accept unique_function
- Issue #2166[4647] - Simple receive_buffer example
- PR #2165[4648] - Fix wait all
- PR #2164[4649] - Fix wait all
- PR #2163[4650] - Fix some typos in config tests
- PR #2162[4651] - Improve #includes
- PR #2160[4652] - Add inspect check for missing #include <list>
- PR #2159[4653] - Add missing finalize call to stop test hanging
- PR #2158[4654] - Algo fixes
- PR #2157[4655] - Stack check
- Issue #2156[4656] - OSX reports stack space incorrectly (generic context coroutines)
- Issue #2155[4657] - Race condition suspected in runtime
- PR #2154[4658] - Replace boost::detail::atomic_count with the new util::atomic_count
- PR #2153[4659] - Fix stack overflow on OSX
- PR #2152[4660] - Define is_bitwise_serializable as is_trivially_copyable when available
- PR #2151[4661] - Adding missing <cstring> for std::mem* functions
- Issue #2150[4662] - Unable to use component clients as action return types

4640 https://github.com/STEllAR-GROUP/hpx/pull/2175
4641 https://github.com/STEllAR-GROUP/hpx/pull/2174
4642 https://github.com/STEllAR-GROUP/hpx/issues/2172
4643 https://github.com/STEllAR-GROUP/hpx/issues/2171
4644 https://github.com/STEllAR-GROUP/hpx/issues/2170
4645 https://github.com/STEllAR-GROUP/hpx/pull/2168
4646 https://github.com/STEllAR-GROUP/hpx/issues/2167
4647 https://github.com/STEllAR-GROUP/hpx/issues/2166
4648 https://github.com/STEllAR-GROUP/hpx/pull/2165
4649 https://github.com/STEllAR-GROUP/hpx/pull/2164
4650 https://github.com/STEllAR-GROUP/hpx/pull/2163
4651 https://github.com/STEllAR-GROUP/hpx/pull/2162
4652 https://github.com/STEllAR-GROUP/hpx/pull/2160
4653 https://github.com/STEllAR-GROUP/hpx/pull/2159
4654 https://github.com/STEllAR-GROUP/hpx/pull/2158
4655 https://github.com/STEllAR-GROUP/hpx/pull/2157
4656 https://github.com/STEllAR-GROUP/hpx/issues/2156
4657 https://github.com/STEllAR-GROUP/hpx/issues/2155
4658 https://github.com/STEllAR-GROUP/hpx/pull/2154
4659 https://github.com/STEllAR-GROUP/hpx/pull/2153
4660 https://github.com/STEllAR-GROUP/hpx/pull/2152
4661 https://github.com/STEllAR-GROUP/hpx/pull/2151
4662 https://github.com/STEllAR-GROUP/hpx/issues/2150

- PR #2149[4663] - std::memmove copies bytes, use bytes*sizeof(type) when copying larger types
- PR #2146[4664] - Adding customization point for parallel copy/move
- PR #2145[4665] - Applying changes to address warnings issued by latest version of PVS Studio
- Issue #2148[4666] - hpx::parallel::copy is broken after trivially copyable changes
- PR #2144[4667] - Some minor tweaks to compute prototype
- PR #2143[4668] - Added Boost version support information over OSX platform
- PR #2142[4669] - Fixing memory leak in example
- PR #2141[4670] - Add missing specializations in execution policies
- PR #2139[4671] - This PR fixes a few problems reported by Clang's Undefined Behavior sanitizer
- PR #2138[4672] - Revert "Adding fedora docs"
- PR #2136[4673] - Removed double semicolon
- PR #2135[4674] - Add deprecated #include check for hpx_fwd.hpp
- PR #2134[4675] - Resolved memory leak in stencil_8
- PR #2133[4676] - Replace uses of boost pointer containers
- PR #2132[4677] - Removing unused typedef
- PR #2131[4678] - Add several include checks for std facilities
- PR #2130[4679] - Fixing parcel compression, adding test
- PR #2129[4680] - Fix invalid attribute warnings
- Issue #2128[4681] - hpx::init seems to segfault
- PR #2127[4682] - Making executor_traits N-nary
- PR #2126[4683] - GCC 4.6 fails to deduce the correct type in lambda
- PR #2125[4684] - Making parcel coalescing test actually test something
- Issue #2124[4685] - Make a testcase for parcel compression

4663 https://github.com/STEllAR-GROUP/hpx/pull/2149
4664 https://github.com/STEllAR-GROUP/hpx/pull/2146
4665 https://github.com/STEllAR-GROUP/hpx/pull/2145
4666 https://github.com/STEllAR-GROUP/hpx/issues/2148
4667 https://github.com/STEllAR-GROUP/hpx/pull/2144
4668 https://github.com/STEllAR-GROUP/hpx/pull/2143
4669 https://github.com/STEllAR-GROUP/hpx/pull/2142
4670 https://github.com/STEllAR-GROUP/hpx/pull/2141
4671 https://github.com/STEllAR-GROUP/hpx/pull/2139
4672 https://github.com/STEllAR-GROUP/hpx/pull/2138
4673 https://github.com/STEllAR-GROUP/hpx/pull/2136
4674 https://github.com/STEllAR-GROUP/hpx/pull/2135
4675 https://github.com/STEllAR-GROUP/hpx/pull/2134
4676 https://github.com/STEllAR-GROUP/hpx/pull/2133
4677 https://github.com/STEllAR-GROUP/hpx/pull/2132
4678 https://github.com/STEllAR-GROUP/hpx/pull/2131
4679 https://github.com/STEllAR-GROUP/hpx/pull/2130
4680 https://github.com/STEllAR-GROUP/hpx/pull/2129
4681 https://github.com/STEllAR-GROUP/hpx/issues/2128
4682 https://github.com/STEllAR-GROUP/hpx/pull/2127
4683 https://github.com/STEllAR-GROUP/hpx/pull/2126
4684 https://github.com/STEllAR-GROUP/hpx/pull/2125
4685 https://github.com/STEllAR-GROUP/hpx/issues/2124

- Issue #2123[4686] - hpx/hpx/runtime/applier_fwd.hpp - Multiple defined types
- Issue #2122[4687] - Exception in primary_namespace::resolve_free_list
- Issue #2121[4688] - Possible memory leak in 1d_stencil_8
- PR #2120[4689] - Fixing 2119
- Issue #2119[4690] - reduce_by_key compilation problems
- Issue #2118[4691] - Premature unwrapping of boost::ref'ed arguments
- PR #2117[4692] - Added missing initializer on last constructor for thread_description
- PR #2116[4693] - Use a lightweight bind implementation when no placeholders are given
- PR #2115[4694] - Replace boost::shared_ptr with std::shared_ptr
- PR #2114[4695] - Adding hook functions for executor_parameter_traits supporting timers
- Issue #2113[4696] - Compilation error with gcc version 4.9.3 (MacPorts gcc49 4.9.3_0)
- PR #2112[4697] - Replace uses of safe_bool with explicit operator bool
- Issue #2111[4698] - Compilation error on QT example
- Issue #2110[4699] - Compilation error when passing non-future argument to unwrapped continuation in dataflow
- Issue #2109[4700] - Warning while compiling hpx
- Issue #2109[4701] - Stack trace of last bug causing issues with octotiger
- Issue #2108[4702] - Stack trace of last bug causing issues with octotiger
- PR #2107[4703] - Making sure that a missing parcel_coalescing module does not cause startup exceptions
- PR #2106[4704] - Stop using hpx_fwd.hpp
- Issue #2105[4705] - coalescing plugin handler is not optional any more
- Issue #2104[4706] - Make executor_traits N-nary
- Issue #2103[4707] - Build error with octotiger and hpx commit e657426d
- PR #2102[4708] - Combining thread data storage

[4686] https://github.com/STEllAR-GROUP/hpx/issues/2123
[4687] https://github.com/STEllAR-GROUP/hpx/issues/2122
[4688] https://github.com/STEllAR-GROUP/hpx/issues/2121
[4689] https://github.com/STEllAR-GROUP/hpx/pull/2120
[4690] https://github.com/STEllAR-GROUP/hpx/issues/2119
[4691] https://github.com/STEllAR-GROUP/hpx/issues/2118
[4692] https://github.com/STEllAR-GROUP/hpx/pull/2117
[4693] https://github.com/STEllAR-GROUP/hpx/pull/2116
[4694] https://github.com/STEllAR-GROUP/hpx/pull/2115
[4695] https://github.com/STEllAR-GROUP/hpx/pull/2114
[4696] https://github.com/STEllAR-GROUP/hpx/issues/2113
[4697] https://github.com/STEllAR-GROUP/hpx/pull/2112
[4698] https://github.com/STEllAR-GROUP/hpx/issues/2111
[4699] https://github.com/STEllAR-GROUP/hpx/issues/2110
[4700] https://github.com/STEllAR-GROUP/hpx/issues/2109
[4701] https://github.com/STEllAR-GROUP/hpx/issues/2109
[4702] https://github.com/STEllAR-GROUP/hpx/issues/2108
[4703] https://github.com/STEllAR-GROUP/hpx/pull/2107
[4704] https://github.com/STEllAR-GROUP/hpx/pull/2106
[4705] https://github.com/STEllAR-GROUP/hpx/issues/2105
[4706] https://github.com/STEllAR-GROUP/hpx/issues/2104
[4707] https://github.com/STEllAR-GROUP/hpx/issues/2103
[4708] https://github.com/STEllAR-GROUP/hpx/pull/2102

- PR #2101[4709] - Added repartition version of 1d stencil that uses any performance counter
- PR #2100[4710] - Drop obsolete TR1 result_of protocol
- PR #2099[4711] - Replace uses of boost::bind with util::bind
- PR #2098[4712] - Deprecated inspect checks
- PR #2097[4713] - Reduce by key, extends #1141
- PR #2096[4714] - Moving local cache from external to hpx/util
- PR #2095[4715] - Bump minimum required Boost to 1.50.0
- PR #2094[4716] - Add include checks for several Boost utilities
- Issue #2093[4717] - /…/local_cache.hpp(89): error #303: explicit type is missing ("int" assumed)
- PR #2091[4718] - Fix for Raspberry pi build
- PR #2090[4719] - Fix storage size for util::function<>
- PR #2089[4720] - Fix #2088
- Issue #2088[4721] - More verbose output from cmake configuration
- PR #2087[4722] - Making sure init_globally always executes hpx_main
- Issue #2086[4723] - Race condition with recent HPX
- PR #2085[4724] - Adding #include checker
- PR #2084[4725] - Replace boost lock types with standard library ones
- PR #2083[4726] - Simplify packaged task
- PR #2082[4727] - Updating APEX version for testing
- PR #2081[4728] - Cleanup exception headers
- PR #2080[4729] - Make call_once variadic
- Issue #2079[4730] - With GNU C++, line 85 of hpx/config/version.hpp causes link failure when linking application
- Issue #2078[4731] - Simple test fails with _GLIBCXX_DEBUG defined

4709 https://github.com/STEllAR-GROUP/hpx/pull/2101
4710 https://github.com/STEllAR-GROUP/hpx/pull/2100
4711 https://github.com/STEllAR-GROUP/hpx/pull/2099
4712 https://github.com/STEllAR-GROUP/hpx/pull/2098
4713 https://github.com/STEllAR-GROUP/hpx/pull/2097
4714 https://github.com/STEllAR-GROUP/hpx/pull/2096
4715 https://github.com/STEllAR-GROUP/hpx/pull/2095
4716 https://github.com/STEllAR-GROUP/hpx/pull/2094
4717 https://github.com/STEllAR-GROUP/hpx/issues/2093
4718 https://github.com/STEllAR-GROUP/hpx/pull/2091
4719 https://github.com/STEllAR-GROUP/hpx/pull/2090
4720 https://github.com/STEllAR-GROUP/hpx/pull/2089
4721 https://github.com/STEllAR-GROUP/hpx/issues/2088
4722 https://github.com/STEllAR-GROUP/hpx/pull/2087
4723 https://github.com/STEllAR-GROUP/hpx/issues/2086
4724 https://github.com/STEllAR-GROUP/hpx/pull/2085
4725 https://github.com/STEllAR-GROUP/hpx/pull/2084
4726 https://github.com/STEllAR-GROUP/hpx/pull/2083
4727 https://github.com/STEllAR-GROUP/hpx/pull/2082
4728 https://github.com/STEllAR-GROUP/hpx/pull/2081
4729 https://github.com/STEllAR-GROUP/hpx/pull/2080
4730 https://github.com/STEllAR-GROUP/hpx/issues/2079
4731 https://github.com/STEllAR-GROUP/hpx/issues/2078

- PR #2077[4732] - Instantiate board in nqueen client
- PR #2076[4733] - Moving coalescing registration to TUs
- PR #2075[4734] - Fixed some documentation typos
- PR #2074[4735] - Adding flush-mode to message handler flush
- PR #2073[4736] - Fixing performance regression introduced lately
- PR #2072[4737] - Refactor local::condition_variable
- PR #2071[4738] - Timer based on boost::asio::deadline_timer
- PR #2070[4739] - Refactor tuple based functionality
- PR #2069[4740] - Fixed typos
- Issue #2068[4741] - Seg fault with octotiger
- PR #2067[4742] - Algorithm cleanup
- PR #2066[4743] - Split credit fixes
- PR #2065[4744] - Rename HPX_MOVABLE_BUT_NOT_COPYABLE to HPX_MOVABLE_ONLY
- PR #2064[4745] - Fixed some typos in docs
- PR #2063[4746] - Adding example demonstrating template components
- Issue #2062[4747] - Support component templates
- PR #2061[4748] - Replace some uses of lexical_cast<string> with C++11 std::to_string
- PR #2060[4749] - Replace uses of boost::noncopyable with HPX_NON_COPYABLE
- PR #2059[4750] - Adding missing for_loop algorithms
- PR #2058[4751] - Move several definitions to more appropriate headers
- PR #2057[4752] - Simplify assert_owns_lock and ignore_while_checking
- PR #2056[4753] - Replacing std::result_of with util::result_of
- PR #2055[4754] - Fix process launching/connecting back

[4732] https://github.com/STEllAR-GROUP/hpx/pull/2077
[4733] https://github.com/STEllAR-GROUP/hpx/pull/2076
[4734] https://github.com/STEllAR-GROUP/hpx/pull/2075
[4735] https://github.com/STEllAR-GROUP/hpx/pull/2074
[4736] https://github.com/STEllAR-GROUP/hpx/pull/2073
[4737] https://github.com/STEllAR-GROUP/hpx/pull/2072
[4738] https://github.com/STEllAR-GROUP/hpx/pull/2071
[4739] https://github.com/STEllAR-GROUP/hpx/pull/2070
[4740] https://github.com/STEllAR-GROUP/hpx/pull/2069
[4741] https://github.com/STEllAR-GROUP/hpx/issues/2068
[4742] https://github.com/STEllAR-GROUP/hpx/pull/2067
[4743] https://github.com/STEllAR-GROUP/hpx/pull/2066
[4744] https://github.com/STEllAR-GROUP/hpx/pull/2065
[4745] https://github.com/STEllAR-GROUP/hpx/pull/2064
[4746] https://github.com/STEllAR-GROUP/hpx/pull/2063
[4747] https://github.com/STEllAR-GROUP/hpx/issues/2062
[4748] https://github.com/STEllAR-GROUP/hpx/pull/2061
[4749] https://github.com/STEllAR-GROUP/hpx/pull/2060
[4750] https://github.com/STEllAR-GROUP/hpx/pull/2059
[4751] https://github.com/STEllAR-GROUP/hpx/pull/2058
[4752] https://github.com/STEllAR-GROUP/hpx/pull/2057
[4753] https://github.com/STEllAR-GROUP/hpx/pull/2056
[4754] https://github.com/STEllAR-GROUP/hpx/pull/2055

- PR #2054[4755] - Add a forwarding coroutine header
- PR #2053[4756] - Replace uses of boost::unordered_map with std::unordered_map
- PR #2052[4757] - Rewrite tuple unwrap
- PR #2050[4758] - Replace uses of BOOST_SCOPED_ENUM with C++11 scoped enums
- PR #2049[4759] - Attempt to narrow down split_credit problem
- PR #2048[4760] - Fixing gcc startup hangs
- PR #2047[4761] - Fixing when_xxx and wait_xxx for MSVC12
- PR #2046[4762] - adding persistent_auto_chunk_size and related tests for for_each
- PR #2045[4763] - Fixing HPX_HAVE_THREAD_BACKTRACE_DEPTH build time configuration
- PR #2044[4764] - Adding missing service executor types
- PR #2043[4765] - Removing ambiguous definitions for is_future_range and future_range_traits
- PR #2042[4766] - Clarify that HPX builds can use (much) more than 2GB per process
- PR #2041[4767] - Changing future_iterator_traits to support pointers
- Issue #2040[4768] - Improve documentation memory usage warning?
- PR #2039[4769] - Coroutine cleanup
- PR #2038[4770] - Fix cmake policy CMP0042 warning MACOSX_RPATH
- PR #2037[4771] - Avoid redundant specialization of [**unique_**]function_nonser
- PR #2036[4772] - nvcc dies with an internal error upon pushing/popping warnings inside templates
- Issue #2035[4773] - Use a less restrictive iterator definition in hpx::lcos::detail::future_iterator_traits
- PR #2034[4774] - Fixing compilation error with thread queue wait time performance counter
- Issue #2033[4775] - Compilation error when compiling with thread queue waittime performance counter
- Issue #2032[4776] - Ambiguous template instantiation for is_future_range and future_range_traits.
- PR #2031[4777] - Don't restart timer on every incoming parcel

4755 https://github.com/STEllAR-GROUP/hpx/pull/2054
4756 https://github.com/STEllAR-GROUP/hpx/pull/2053
4757 https://github.com/STEllAR-GROUP/hpx/pull/2052
4758 https://github.com/STEllAR-GROUP/hpx/pull/2050
4759 https://github.com/STEllAR-GROUP/hpx/pull/2049
4760 https://github.com/STEllAR-GROUP/hpx/pull/2048
4761 https://github.com/STEllAR-GROUP/hpx/pull/2047
4762 https://github.com/STEllAR-GROUP/hpx/pull/2046
4763 https://github.com/STEllAR-GROUP/hpx/pull/2045
4764 https://github.com/STEllAR-GROUP/hpx/pull/2044
4765 https://github.com/STEllAR-GROUP/hpx/pull/2043
4766 https://github.com/STEllAR-GROUP/hpx/pull/2042
4767 https://github.com/STEllAR-GROUP/hpx/pull/2041
4768 https://github.com/STEllAR-GROUP/hpx/issues/2040
4769 https://github.com/STEllAR-GROUP/hpx/pull/2039
4770 https://github.com/STEllAR-GROUP/hpx/pull/2038
4771 https://github.com/STEllAR-GROUP/hpx/pull/2037
4772 https://github.com/STEllAR-GROUP/hpx/pull/2036
4773 https://github.com/STEllAR-GROUP/hpx/issues/2035
4774 https://github.com/STEllAR-GROUP/hpx/pull/2034
4775 https://github.com/STEllAR-GROUP/hpx/issues/2033
4776 https://github.com/STEllAR-GROUP/hpx/issues/2032
4777 https://github.com/STEllAR-GROUP/hpx/pull/2031

- PR #2030[4778] - Unify handling of execution policies in parallel algorithms
- PR #2029[4779] - Make pkg-config .pc files use .dylib on OSX
- PR #2028[4780] - Adding process component
- PR #2027[4781] - Making check for compiler compatibility independent on compiler path
- PR #2025[4782] - Fixing inspect tool
- PR #2024[4783] - Intel13 removal
- PR #2023[4784] - Fix errors related to older boost versions and parameter pack expansions in lambdas
- Issue #2022[4785] - gmake fail: "No rule to make target /usr/lib46/libboost_context-mt.so"
- PR #2021[4786] - Added Sudoku example
- Issue #2020[4787] - Make errors related to init_globally.cpp example while building HPX out of the box
- PR #2019[4788] - Fixed some compilation and cmake errors encountered in nqueen example
- PR #2018[4789] - For loop algorithms
- PR #2017[4790] - Non-recursive at_index implementation
- Issue #2016[4791] - Add index-based for-loops
- Issue #2015[4792] - Change default bind-mode to balanced
- PR #2014[4793] - Fixed dataflow if invoked action returns a future
- PR #2013[4794] - Fixing compilation issues with external example
- PR #2012[4795] - Added Sierpinski Triangle example
- Issue #2011[4796] - Compilation error while running sample hello_world_component code
- PR #2010[4797] - Segmented move implemented for hpx::vector
- Issue #2009[4798] - pkg-config order incorrect on 14.04 / GCC 4.8
- Issue #2008[4799] - Compilation error in dataflow of action returning a future
- PR #2007[4800] - Adding new performance counter exposing overall scheduler time

[4778] https://github.com/STEllAR-GROUP/hpx/pull/2030
[4779] https://github.com/STEllAR-GROUP/hpx/pull/2029
[4780] https://github.com/STEllAR-GROUP/hpx/pull/2028
[4781] https://github.com/STEllAR-GROUP/hpx/pull/2027
[4782] https://github.com/STEllAR-GROUP/hpx/pull/2025
[4783] https://github.com/STEllAR-GROUP/hpx/pull/2024
[4784] https://github.com/STEllAR-GROUP/hpx/pull/2023
[4785] https://github.com/STEllAR-GROUP/hpx/issues/2022
[4786] https://github.com/STEllAR-GROUP/hpx/pull/2021
[4787] https://github.com/STEllAR-GROUP/hpx/issues/2020
[4788] https://github.com/STEllAR-GROUP/hpx/pull/2019
[4789] https://github.com/STEllAR-GROUP/hpx/pull/2018
[4790] https://github.com/STEllAR-GROUP/hpx/pull/2017
[4791] https://github.com/STEllAR-GROUP/hpx/issues/2016
[4792] https://github.com/STEllAR-GROUP/hpx/issues/2015
[4793] https://github.com/STEllAR-GROUP/hpx/pull/2014
[4794] https://github.com/STEllAR-GROUP/hpx/pull/2013
[4795] https://github.com/STEllAR-GROUP/hpx/pull/2012
[4796] https://github.com/STEllAR-GROUP/hpx/issues/2011
[4797] https://github.com/STEllAR-GROUP/hpx/pull/2010
[4798] https://github.com/STEllAR-GROUP/hpx/issues/2009
[4799] https://github.com/STEllAR-GROUP/hpx/issues/2008
[4800] https://github.com/STEllAR-GROUP/hpx/pull/2007

- PR #2006[4801] - Function includes
- PR #2005[4802] - Adding an example demonstrating how to initialize HPX from a global object
- PR #2004[4803] - Fixing 2000
- PR #2003[4804] - Adding generation parameter to gather to enable using it more than once
- PR #2002[4805] - Turn on position independent code to solve link problem with hpx_init
- Issue #2001[4806] - Gathering more than once segfaults
- Issue #2000[4807] - Undefined reference to hpx::assertion_failed
- Issue #1999[4808] - Seg fault in hpx::lcos::base_lco_with_value<*>::set_value_nonvirt() when running octo-tiger
- PR #1998[4809] - Detect unknown command line options
- PR #1997[4810] - Extending thread description
- PR #1996[4811] - Adding natvis files to solution (MSVC only)
- Issue #1995[4812] - Command line handling does not produce error
- PR #1994[4813] - Possible missing include in test_utils.hpp
- PR #1993[4814] - Add missing LANGUAGES tag to a hpx_add_compile_flag_if_available() call in CMakeLists.txt
- PR #1992[4815] - Fixing shared_executor_test
- PR #1991[4816] - Making sure the winsock library is properly initialized
- PR #1990[4817] - Fixing bind_test placeholder ambiguity coming from boost-1.60
- PR #1989[4818] - Performance tuning
- PR #1987[4819] - Make configurable size of internal storage in util::function
- PR #1986[4820] - AGAS Refactoring+1753 Cache mods
- PR #1985[4821] - Adding missing task_block::run() overload taking an executor
- PR #1984[4822] - Adding an optimized LRU Cache implementation (for AGAS)
- PR #1983[4823] - Avoid invoking migration table look up for all objects

[4801] https://github.com/STEllAR-GROUP/hpx/pull/2006
[4802] https://github.com/STEllAR-GROUP/hpx/pull/2005
[4803] https://github.com/STEllAR-GROUP/hpx/pull/2004
[4804] https://github.com/STEllAR-GROUP/hpx/pull/2003
[4805] https://github.com/STEllAR-GROUP/hpx/pull/2002
[4806] https://github.com/STEllAR-GROUP/hpx/issues/2001
[4807] https://github.com/STEllAR-GROUP/hpx/issues/2000
[4808] https://github.com/STEllAR-GROUP/hpx/issues/1999
[4809] https://github.com/STEllAR-GROUP/hpx/pull/1998
[4810] https://github.com/STEllAR-GROUP/hpx/pull/1997
[4811] https://github.com/STEllAR-GROUP/hpx/pull/1996
[4812] https://github.com/STEllAR-GROUP/hpx/issues/1995
[4813] https://github.com/STEllAR-GROUP/hpx/pull/1994
[4814] https://github.com/STEllAR-GROUP/hpx/pull/1993
[4815] https://github.com/STEllAR-GROUP/hpx/pull/1992
[4816] https://github.com/STEllAR-GROUP/hpx/pull/1991
[4817] https://github.com/STEllAR-GROUP/hpx/pull/1990
[4818] https://github.com/STEllAR-GROUP/hpx/pull/1989
[4819] https://github.com/STEllAR-GROUP/hpx/pull/1987
[4820] https://github.com/STEllAR-GROUP/hpx/pull/1986
[4821] https://github.com/STEllAR-GROUP/hpx/pull/1985
[4822] https://github.com/STEllAR-GROUP/hpx/pull/1984
[4823] https://github.com/STEllAR-GROUP/hpx/pull/1983

- PR #1981[4824] - Replacing uintptr_t (which is not defined everywhere) with std::size_t
- PR #1980[4825] - Optimizing LCO continuations
- PR #1979[4826] - Fixing Cori
- PR #1978[4827] - Fix test check that got broken in hasty fix to memory overflow
- PR #1977[4828] - Refactor action traits
- PR #1976[4829] - Fixes typo in README.rst
- PR #1975[4830] - Reduce size of benchmark timing arrays to fix test failures
- PR #1974[4831] - Add action to update data owned by the partitioned_vector component
- PR #1972[4832] - Adding partitioned_vector SPMD example
- PR #1971[4833] - Fixing 1965
- PR #1970[4834] - Papi fixes
- PR #1969[4835] - Fixing continuation recursions to not depend on fixed amount of recursions
- PR #1968[4836] - More segmented algorithms
- Issue #1967[4837] - Simplify component implementations
- PR #1966[4838] - Migrate components
- Issue #1964[4839] - fatal error: 'boost/lockfree/detail/branch_hints.hpp' file not found
- Issue #1962[4840] - parallel:copy_if has race condition when used on in place arrays
- PR #1963[4841] - Fixing Static Parcelport initialization
- PR #1961[4842] - Fix function target
- Issue #1960[4843] - Papi counters don't reset
- PR #1959[4844] - Fixing 1958
- Issue #1958[4845] - inclusive_scan gives incorrect results with non-commutative operator
- PR #1957[4846] - Fixing #1950

4824 https://github.com/STEllAR-GROUP/hpx/pull/1981
4825 https://github.com/STEllAR-GROUP/hpx/pull/1980
4826 https://github.com/STEllAR-GROUP/hpx/pull/1979
4827 https://github.com/STEllAR-GROUP/hpx/pull/1978
4828 https://github.com/STEllAR-GROUP/hpx/pull/1977
4829 https://github.com/STEllAR-GROUP/hpx/pull/1976
4830 https://github.com/STEllAR-GROUP/hpx/pull/1975
4831 https://github.com/STEllAR-GROUP/hpx/pull/1974
4832 https://github.com/STEllAR-GROUP/hpx/pull/1972
4833 https://github.com/STEllAR-GROUP/hpx/pull/1971
4834 https://github.com/STEllAR-GROUP/hpx/pull/1970
4835 https://github.com/STEllAR-GROUP/hpx/pull/1969
4836 https://github.com/STEllAR-GROUP/hpx/pull/1968
4837 https://github.com/STEllAR-GROUP/hpx/issues/1967
4838 https://github.com/STEllAR-GROUP/hpx/pull/1966
4839 https://github.com/STEllAR-GROUP/hpx/issues/1964
4840 https://github.com/STEllAR-GROUP/hpx/issues/1962
4841 https://github.com/STEllAR-GROUP/hpx/pull/1963
4842 https://github.com/STEllAR-GROUP/hpx/pull/1961
4843 https://github.com/STEllAR-GROUP/hpx/issues/1960
4844 https://github.com/STEllAR-GROUP/hpx/pull/1959
4845 https://github.com/STEllAR-GROUP/hpx/issues/1958
4846 https://github.com/STEllAR-GROUP/hpx/pull/1957

- PR #1956[4847] - Sort by key example
- PR #1955[4848] - Adding regression test for #1946: Hang in wait_all() in distributed run
- Issue #1954[4849] - HPX releases should not use -Werror
- PR #1953[4850] - Adding performance analysis for AGAS cache
- PR #1952[4851] - Adapting test for explicit variadics to fail for gcc 4.6
- PR #1951[4852] - Fixing memory leak
- Issue #1950[4853] - Simplify external builds
- PR #1949[4854] - Fixing yet another lock that is being held during suspension
- PR #1948[4855] - Fixed container algorithms for Intel
- PR #1947[4856] - Adding workaround for tagged_tuple
- Issue #1946[4857] - Hang in wait_all() in distributed run
- PR #1945[4858] - Fixed container algorithm tests
- Issue #1944[4859] - assertion 'p.destination_locality() == hpx::get_locality()' failed
- PR #1943[4860] - Fix a couple of compile errors with clang
- PR #1942[4861] - Making parcel coalescing functional
- Issue #1941[4862] - Re-enable parcel coalescing
- PR #1940[4863] - Touching up make_future
- PR #1939[4864] - Fixing problems in over-subscription management in the resource manager
- PR #1938[4865] - Removing use of unified Boost.Thread header
- PR #1937[4866] - Cleaning up the use of Boost.Accumulator headers
- PR #1936[4867] - Making sure interval timer is started for aggregating performance counters
- PR #1935[4868] - Tagged results
- PR #1934[4869] - Fix remote async with deferred launch policy

4847 https://github.com/STEllAR-GROUP/hpx/pull/1956
4848 https://github.com/STEllAR-GROUP/hpx/pull/1955
4849 https://github.com/STEllAR-GROUP/hpx/issues/1954
4850 https://github.com/STEllAR-GROUP/hpx/pull/1953
4851 https://github.com/STEllAR-GROUP/hpx/pull/1952
4852 https://github.com/STEllAR-GROUP/hpx/pull/1951
4853 https://github.com/STEllAR-GROUP/hpx/issues/1950
4854 https://github.com/STEllAR-GROUP/hpx/pull/1949
4855 https://github.com/STEllAR-GROUP/hpx/pull/1948
4856 https://github.com/STEllAR-GROUP/hpx/pull/1947
4857 https://github.com/STEllAR-GROUP/hpx/issues/1946
4858 https://github.com/STEllAR-GROUP/hpx/pull/1945
4859 https://github.com/STEllAR-GROUP/hpx/issues/1944
4860 https://github.com/STEllAR-GROUP/hpx/pull/1943
4861 https://github.com/STEllAR-GROUP/hpx/pull/1942
4862 https://github.com/STEllAR-GROUP/hpx/issues/1941
4863 https://github.com/STEllAR-GROUP/hpx/pull/1940
4864 https://github.com/STEllAR-GROUP/hpx/pull/1939
4865 https://github.com/STEllAR-GROUP/hpx/pull/1938
4866 https://github.com/STEllAR-GROUP/hpx/pull/1937
4867 https://github.com/STEllAR-GROUP/hpx/pull/1936
4868 https://github.com/STEllAR-GROUP/hpx/pull/1935
4869 https://github.com/STEllAR-GROUP/hpx/pull/1934

- Issue #1933[4870] - Floating point exception in `statistics_counter<boost::accumulators::tag::mean>::get_counter_v`
- PR #1932[4871] - Removing superfluous includes of boost/lockfree/detail/branch_hints.hpp
- PR #1931[4872] - fix compilation with clang 3.8.0
- Issue #1930[4873] - Missing online documentation for HPX 0.9.11
- PR #1929[4874] - LWG2485: get() should be overloaded for const tuple&&
- PR #1928[4875] - Revert "Using ninja for circle-ci builds"
- PR #1927[4876] - Using ninja for circle-ci builds
- PR #1926[4877] - Fixing serialization of std::array
- Issue #1925[4878] - Issues with static HPX libraries
- Issue #1924[4879] - Performance degrading over time
- Issue #1923[4880] - serialization of std::array appears broken in latest commit
- PR #1922[4881] - Container algorithms
- PR #1921[4882] - Tons of smaller quality improvements
- Issue #1920[4883] - Seg fault in hpx::serialization::output_archive::add_gid when running octotiger
- Issue #1919[4884] - Intel 15 compiler bug preventing HPX build
- PR #1918[4885] - Address sanitizer fixes
- PR #1917[4886] - Fixing compilation problems of parallel::sort with Intel compilers
- PR #1916[4887] - Making sure code compiles if HPX_WITH_HWLOC=Off
- Issue #1915[4888] - max_cores undefined if HPX_WITH_HWLOC=Off
- PR #1913[4889] - Add utility member functions for partitioned_vector
- PR #1912[4890] - Adding support for invoking actions to dataflow
- PR #1911[4891] - Adding first batch of container algorithms
- PR #1910[4892] - Keep cmake_module_path

4870 https://github.com/STEllAR-GROUP/hpx/issues/1933
4871 https://github.com/STEllAR-GROUP/hpx/pull/1932
4872 https://github.com/STEllAR-GROUP/hpx/pull/1931
4873 https://github.com/STEllAR-GROUP/hpx/issues/1930
4874 https://github.com/STEllAR-GROUP/hpx/pull/1929
4875 https://github.com/STEllAR-GROUP/hpx/pull/1928
4876 https://github.com/STEllAR-GROUP/hpx/pull/1927
4877 https://github.com/STEllAR-GROUP/hpx/pull/1926
4878 https://github.com/STEllAR-GROUP/hpx/issues/1925
4879 https://github.com/STEllAR-GROUP/hpx/issues/1924
4880 https://github.com/STEllAR-GROUP/hpx/issues/1923
4881 https://github.com/STEllAR-GROUP/hpx/pull/1922
4882 https://github.com/STEllAR-GROUP/hpx/pull/1921
4883 https://github.com/STEllAR-GROUP/hpx/issues/1920
4884 https://github.com/STEllAR-GROUP/hpx/issues/1919
4885 https://github.com/STEllAR-GROUP/hpx/pull/1918
4886 https://github.com/STEllAR-GROUP/hpx/pull/1917
4887 https://github.com/STEllAR-GROUP/hpx/pull/1916
4888 https://github.com/STEllAR-GROUP/hpx/issues/1915
4889 https://github.com/STEllAR-GROUP/hpx/pull/1913
4890 https://github.com/STEllAR-GROUP/hpx/pull/1912
4891 https://github.com/STEllAR-GROUP/hpx/pull/1911
4892 https://github.com/STEllAR-GROUP/hpx/pull/1910

- PR #1909[4893] - Fix mpirun with pbs
- PR #1908[4894] - Changing parallel::sort to return the last iterator as proposed by N4560
- PR #1907[4895] - Adding a minimum version for Open MPI
- PR #1906[4896] - Updates to the Release Procedure
- PR #1905[4897] - Fixing #1903
- PR #1904[4898] - Making sure std containers are cleared before serialization loads data
- Issue #1903[4899] - When running octotiger, I get: assertion `'(*new_gids_)[gid].size() == 1' failed: HPX(assertion_failure)`
- Issue #1902[4900] - Immediate crash when running hpx/octotiger with _GLIBCXX_DEBUG defined.
- PR #1901[4901] - Making non-serializable classes non-serializable
- Issue #1900[4902] - Two possible issues with std::list serialization
- PR #1899[4903] - Fixing a problem with credit splitting as revealed by #1898
- Issue #1898[4904] - Accessing component from locality where it was not created segfaults
- PR #1897[4905] - Changing parallel::sort to return the last iterator as proposed by N4560
- Issue #1896[4906] - version 1.0?
- Issue #1895[4907] - Warning comment on numa_allocator is not very clear
- PR #1894[4908] - Add support for compilers that have thread_local
- PR #1893[4909] - Fixing 1890
- PR #1892[4910] - Adds typed future_type for executor_traits
- PR #1891[4911] - Fix wording in certain parallel algorithm docs
- Issue #1890[4912] - Invoking papi counters give segfault
- PR #1889[4913] - Fixing problems as reported by clang-check
- PR #1888[4914] - WIP parallel is_heap
- PR #1887[4915] - Fixed resetting performance counters related to idle-rate, etc

[4893] https://github.com/STEllAR-GROUP/hpx/pull/1909
[4894] https://github.com/STEllAR-GROUP/hpx/pull/1908
[4895] https://github.com/STEllAR-GROUP/hpx/pull/1907
[4896] https://github.com/STEllAR-GROUP/hpx/pull/1906
[4897] https://github.com/STEllAR-GROUP/hpx/pull/1905
[4898] https://github.com/STEllAR-GROUP/hpx/pull/1904
[4899] https://github.com/STEllAR-GROUP/hpx/issues/1903
[4900] https://github.com/STEllAR-GROUP/hpx/issues/1902
[4901] https://github.com/STEllAR-GROUP/hpx/pull/1901
[4902] https://github.com/STEllAR-GROUP/hpx/issues/1900
[4903] https://github.com/STEllAR-GROUP/hpx/pull/1899
[4904] https://github.com/STEllAR-GROUP/hpx/issues/1898
[4905] https://github.com/STEllAR-GROUP/hpx/pull/1897
[4906] https://github.com/STEllAR-GROUP/hpx/issues/1896
[4907] https://github.com/STEllAR-GROUP/hpx/issues/1895
[4908] https://github.com/STEllAR-GROUP/hpx/pull/1894
[4909] https://github.com/STEllAR-GROUP/hpx/pull/1893
[4910] https://github.com/STEllAR-GROUP/hpx/pull/1892
[4911] https://github.com/STEllAR-GROUP/hpx/pull/1891
[4912] https://github.com/STEllAR-GROUP/hpx/issues/1890
[4913] https://github.com/STEllAR-GROUP/hpx/pull/1889
[4914] https://github.com/STEllAR-GROUP/hpx/pull/1888
[4915] https://github.com/STEllAR-GROUP/hpx/pull/1887

- Issue #1886[4916] - Run hpx with qsub does not work
- PR #1885[4917] - Warning cleaning pass
- PR #1884[4918] - Add missing parallel algorithm header
- PR #1883[4919] - Add feature test for thread_local on Clang for TLS
- PR #1882[4920] - Fix some redundant qualifiers
- Issue #1881[4921] - Unable to compile Octotiger using HPX and Intel MPI on SuperMIC
- Issue #1880[4922] - clang with libc++ on Linux needs TLS case
- PR #1879[4923] - Doc fixes for #1868
- PR #1878[4924] - Simplify functions
- PR #1877[4925] - Removing most usage of Boost.Config
- PR #1876[4926] - Add missing parallel algorithms to algorithm.hpp
- PR #1875[4927] - Simplify callables
- PR #1874[4928] - Address long standing FIXME on using `std::unique_ptr` with incomplete types
- PR #1873[4929] - Fixing 1871
- PR #1872[4930] - Making sure PBS environment uses specified node list even if no PBS_NODEFILE env is available
- Issue #1871[4931] - Fortran checks should be optional
- PR #1870[4932] - Touch local::mutex
- PR #1869[4933] - Documentation refactoring based off #1868
- PR #1867[4934] - Embrace static_assert
- PR #1866[4935] - Fix #1803 with documentation refactoring
- PR #1865[4936] - Setting OUTPUT_NAME as target properties
- PR #1863[4937] - Use SYSTEM for boost includes
- PR #1862[4938] - Minor cleanups

[4916] https://github.com/STEllAR-GROUP/hpx/issues/1886
[4917] https://github.com/STEllAR-GROUP/hpx/pull/1885
[4918] https://github.com/STEllAR-GROUP/hpx/pull/1884
[4919] https://github.com/STEllAR-GROUP/hpx/pull/1883
[4920] https://github.com/STEllAR-GROUP/hpx/pull/1882
[4921] https://github.com/STEllAR-GROUP/hpx/issues/1881
[4922] https://github.com/STEllAR-GROUP/hpx/issues/1880
[4923] https://github.com/STEllAR-GROUP/hpx/pull/1879
[4924] https://github.com/STEllAR-GROUP/hpx/pull/1878
[4925] https://github.com/STEllAR-GROUP/hpx/pull/1877
[4926] https://github.com/STEllAR-GROUP/hpx/pull/1876
[4927] https://github.com/STEllAR-GROUP/hpx/pull/1875
[4928] https://github.com/STEllAR-GROUP/hpx/pull/1874
[4929] https://github.com/STEllAR-GROUP/hpx/pull/1873
[4930] https://github.com/STEllAR-GROUP/hpx/pull/1872
[4931] https://github.com/STEllAR-GROUP/hpx/issues/1871
[4932] https://github.com/STEllAR-GROUP/hpx/pull/1870
[4933] https://github.com/STEllAR-GROUP/hpx/pull/1869
[4934] https://github.com/STEllAR-GROUP/hpx/pull/1867
[4935] https://github.com/STEllAR-GROUP/hpx/pull/1866
[4936] https://github.com/STEllAR-GROUP/hpx/pull/1865
[4937] https://github.com/STEllAR-GROUP/hpx/pull/1863
[4938] https://github.com/STEllAR-GROUP/hpx/pull/1862

- PR #1861[4939] - Minor Corrections for Release
- PR #1860[4940] - Fixing hpx gdb script
- Issue #1859[4941] - reset_active_counters resets times and thread counts before some of the counters are evaluated
- PR #1858[4942] - Release V0.9.11
- PR #1857[4943] - removing diskperf example from 9.11 release
- PR #1856[4944] - fix return in packaged_task_base::reset()
- Issue #1842[4945] - Install error: file INSTALL cannot find libhpx_parcel_coalescing.so.0.9.11
- PR #1839[4946] - Adding fedora docs
- PR #1824[4947] - Changing version on master to V0.9.12
- PR #1818[4948] - Fixing #1748
- Issue #1815[4949] - seg fault in AGAS
- Issue #1803[4950] - wait_all documentation
- Issue #1796[4951] - Outdated documentation to be revised
- Issue #1759[4952] - glibc munmap_chunk or free(): invalid pointer on SuperMIC
- Issue #1753[4953] - HPX performance degrades with time since execution begins
- Issue #1748[4954] - All public HPX headers need to be self contained
- PR #1719[4955] - How to build HPX with Visual Studio
- Issue #1684[4956] - Race condition when using –hpx:connect?
- PR #1658[4957] - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1641[4958] - Generic client
- Issue #1632[4959] - heartbeat example fails on separate nodes
- PR #1603[4960] - Adds preferred namespace check to inspect tool
- Issue #1559[4961] - Extend inspect tool

4939 https://github.com/STEllAR-GROUP/hpx/pull/1861
4940 https://github.com/STEllAR-GROUP/hpx/pull/1860
4941 https://github.com/STEllAR-GROUP/hpx/issues/1859
4942 https://github.com/STEllAR-GROUP/hpx/pull/1858
4943 https://github.com/STEllAR-GROUP/hpx/pull/1857
4944 https://github.com/STEllAR-GROUP/hpx/pull/1856
4945 https://github.com/STEllAR-GROUP/hpx/issues/1842
4946 https://github.com/STEllAR-GROUP/hpx/pull/1839
4947 https://github.com/STEllAR-GROUP/hpx/pull/1824
4948 https://github.com/STEllAR-GROUP/hpx/pull/1818
4949 https://github.com/STEllAR-GROUP/hpx/issues/1815
4950 https://github.com/STEllAR-GROUP/hpx/issues/1803
4951 https://github.com/STEllAR-GROUP/hpx/issues/1796
4952 https://github.com/STEllAR-GROUP/hpx/issues/1759
4953 https://github.com/STEllAR-GROUP/hpx/issues/1753
4954 https://github.com/STEllAR-GROUP/hpx/issues/1748
4955 https://github.com/STEllAR-GROUP/hpx/pull/1719
4956 https://github.com/STEllAR-GROUP/hpx/issues/1684
4957 https://github.com/STEllAR-GROUP/hpx/pull/1658
4958 https://github.com/STEllAR-GROUP/hpx/pull/1641
4959 https://github.com/STEllAR-GROUP/hpx/issues/1632
4960 https://github.com/STEllAR-GROUP/hpx/pull/1603
4961 https://github.com/STEllAR-GROUP/hpx/issues/1559

- Issue #1523[4962] - Remote async with deferred launch policy never executes
- Issue #1472[4963] - Serialization issues
- Issue #1457[4964] - Implement N4392: C++ Latches and Barriers
- PR #1444[4965] - Enabling usage of moveonly types for component construction
- Issue #1407[4966] - The Intel 13 compiler has failing unit tests
- Issue #1405[4967] - Allow component constructors to take movable only types
- Issue #1265[4968] - Enable dataflow() to be usable with actions
- Issue #1236[4969] - NUMA aware allocators
- Issue #802[4970] - Fix Broken Examples
- Issue #559[4971] - Add hpx::migrate facility
- Issue #449[4972] - Make actions with template arguments usable and add documentation
- Issue #279[4973] - Refactor addressing_service into a base class and two derived classes
- Issue #224[4974] - Changing thread state metadata is not thread safe
- Issue #55[4975] - Uniform syntax for enums should be implemented

HPX V0.9.11 (Nov 11, 2015)

Our main focus for this release was the design and development of a coherent set of higher-level APIs exposing various types of parallelism to the application programmer. We introduced the concepts of an `executor`, which can be used to customize the `where` and `when` of execution of tasks in the context of parallelizing codes. We extended all APIs related to managing parallel tasks to support executors which gives the user the choce of either using one of the predefined executor types or to provide its own, possibly application specific, executor. We paid very close attention to align all of these changes with the existing C++ Standards documents or with the ongoing proposals for standardization.

This release is the first after our change to a new development policy. We switched all development to be strictly performed on branches only, all direct commits to our main branch (`master`) are prohibited. Any change has to go through a peer review before it will be merged to `master`. As a result the overall stability of our code base has significantly increased, the development process itself has been simplified. This change manifests itself in a large number of pull-requests which have been merged (please see below for a full list of closed issues and pull-requests). All in all for this release, we closed almost 100 issues and merged over 290 pull-requests. There have been over 1600 commits to the master branch since the last release.

[4962] https://github.com/STEllAR-GROUP/hpx/issues/1523
[4963] https://github.com/STEllAR-GROUP/hpx/issues/1472
[4964] https://github.com/STEllAR-GROUP/hpx/issues/1457
[4965] https://github.com/STEllAR-GROUP/hpx/pull/1444
[4966] https://github.com/STEllAR-GROUP/hpx/issues/1407
[4967] https://github.com/STEllAR-GROUP/hpx/issues/1405
[4968] https://github.com/STEllAR-GROUP/hpx/issues/1265
[4969] https://github.com/STEllAR-GROUP/hpx/issues/1236
[4970] https://github.com/STEllAR-GROUP/hpx/issues/802
[4971] https://github.com/STEllAR-GROUP/hpx/issues/559
[4972] https://github.com/STEllAR-GROUP/hpx/issues/449
[4973] https://github.com/STEllAR-GROUP/hpx/issues/279
[4974] https://github.com/STEllAR-GROUP/hpx/issues/224
[4975] https://github.com/STEllAR-GROUP/hpx/issues/55

General changes

- We are moving into the direction of unifying managed and simple components. As such, the classes `hpx::components::component` and `hpx::components::component_base` have been added which currently just forward to the currently existing simple component facilities. The examples have been converted to only use those two classes.
- Added integration with the CircleCI[4976] hosted continuous integration service. This gives us constant and immediate feedback on the health of our master branch.
- The compiler configuration subsystem in the build system has been reimplemented. Instead of using Boost.Config we now use our own lightweight set of cmake scripts to determine the available language and library features supported by the used compiler.
- The API for creating instances of components has been consolidated. All component instances should be created using the `hpx::new_` only. It allows one to instantiate both, single component instances and multiple component instances. The placement of the created components can be controlled by special distribution policies. Please see the corresponding documentation outlining the use of `hpx::new_`.
- Introduced four new distribution policies which can be used with many API functions which traditionally expected to be used with a locality id. The new distribution policies are:
  - *hpx::components::default_distribution_policy* which tries to place multiple component instances as evenly as possible.
  - *hpx::components::colocating_distribution_policy* which will refer to the locality where a given component instance is currently placed.
  - *hpx::components::binpacking_distribution_policy* which will place multiple component instances as evenly as possible based on any performance counter.
  - *hpx::components::target_distribution_policy* which allows one to represent a given locality in the context of a distrwibution policy.
- The new distribution policies can now be also used with `hpx::async`. This change also deprecates `hpx::async_colocated(id, ...)` which now is replaced by a distribution policy: `hpx::async(hpx::colocated(id), ...)`.
- The `hpx::vector` and `hpx::unordered_map` data structures can now be used with the new distribution policies as well.
- The parallel facility `hpx::parallel::task_region` has been renamed to `hpx::parallel::task_block` based on the changes in the corresponding standardization proposal N4411[4977].
- Added extensions to the parallel facility `hpx::parallel::task_block` allowing to combine a task_block with an execution policy. This implies a minor breaking change as the `hpx::parallel::task_block` is now a template.
- Added new LCOs: `hpx::lcos::latch` and `hpx::lcos::local::latch` which semantically conform to the proposed `std::latch` (see N4399[4978]).
- Added performance counters exposing data related to data transferred by input/output (filesystem) operations (thanks to Maciej Brodowicz).
- Added performance counters allowing to track the number of action invocations (local and remote invocations).
- Added new command line options –hpx:print-counter-at and –hpx:reset-counters.

4976 https://circleci.com/gh/STEllAR-GROUP/hpx
4977 http://wg21.link/n4411
4978 http://wg21.link/n4399

- The `hpx::vector` component has been renamed to `hpx::partitioned_vector` to make it explicit that the underlying memory is not contiguous.
- Introduced a completely new and uniform higher-level parallelism API which is based on executors. All existing parallelism APIs have been adapted to this. We have added a large number of different executor types, such as a numa-aware executor, a this-thread executor, etc.
- Added support for the MingW toolchain on Windows (thanks to Eric Lemanissier).
- HPX now includes support for APEX, (Autonomic Performance Environment for eXascale). APEX is an instrumentation and software adaptation library that provides an interface to TAU profiling / tracing as well as runtime adaptation of HPX applications through policy definitions. For more information and documentation, please see https://github.com/UO-OACISS/xpress-apex. To enable APEX at configuration time, specify `-DHPX_WITH_APEX=On`. To also include support for TAU profiling, specify `-DHPX_WITH_TAU=On` and specify the `-DTAU_ROOT`, `-DTAU_ARCH` and `-DTAU_OPTIONS` cmake parameters.
- We have implemented many more of the *Using parallel algorithms*. Please see Issue #1141[4979] for the list of all available parallel algorithms (thanks to Daniel Bourgeois and John Biddiscombe for contributing their work).

Breaking changes

- We are moving into the direction of unifying managed and simple components. In order to stop exposing the old facilities, all examples have been converted to use the new classes. The breaking change in this release is that performance counters are now a `hpx::components::component_base` instead of `hpx::components::managed_component_base`.
- We removed the support for stackless threads. It turned out that there was no performance benefit when using stackless threads. As such, we decided to clean up our codebase. This feature was not documented.
- The CMake project name has changed from 'hpx' to 'HPX' for consistency and compatibility with naming conventions and other CMake projects. Generated config files go into <prefix>/lib/cmake/HPX and not <prefix>/lib/cmake/hpx.
- The macro `HPX_REGISTER_MINIMAL_COMPONENT_FACTORY` has been deprecated. Please use `HPX_REGISTER_COMPONENT`. instead. The old macro will be removed in the next release.
- The obsolete distributing_factory and binpacking_factory components have been removed. The corresponding functionality is now provided by the `hpx::new_` API function in conjunction with the `hpx::default_layout` and `hpx::binpacking` distribution policies (`hpx::components::default_distribution_policy` and `hpx::components::binpacking_distribution_policy`)
- The API function `hpx::new_colocated` has been deprecated. Please use the consolidated API `hpx::new_` in conjunction with the new `hpx::colocated` distribution policy (`hpx::components::colocating_distribution_policy`) instead. The old API function will still be available for at least one release of *HPX* if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The API function `hpx::async_colocated` has been deprecated. Please use the consolidated API `hpx::async` in conjunction with the new `hpx::colocated` distribution policy (`hpx::components::colocating_distribution_policy`) instead. The old API function will still be available for at least one release of *HPX* if the configuration variable `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` is enabled.
- The obsolete remote_object component has been removed.

[4979] https://github.com/STEllAR-GROUP/hpx/issues/1141

- Replaced the use of Boost.Serialization with our own solution. While the new version is mostly compatible with Boost.Serialization, this change requires some minor code modifications in user code. For more information, please see the corresponding announcement[4980] on the hpx-users@stellar.cct.lsu.edu mailing list.
- The names used by cmake to influence various configuration options have been unified. The new naming scheme relies on all configuration constants to start with `HPX_WITH_...`, while the preprocessor constant which is used at build time starts with `HPX_HAVE_...`. For instance, the former cmake command line `-DHPX_MALLOC=...` now has to be specified a `-DHPX_WITH_MALLOC=...` and will cause the preprocessor constant `HPX_HAVE_MALLOC` to be defined. The actual name of the constant (i.e. `MALLOC`) has not changed. Please see the corresponding documentation for more details (*CMake options*).
- The `get_gid()` functions exposed by the component base classes `hpx::components::server::simple_component_base`, `hpx::components::server::managed_component_base`, and `hpx::components::server::fixed_component_base` have been replaced by two new functions: `get_unmanaged_id()` and `get_id()`. To enable the old function name for backwards compatibility, use the cmake configuration option `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY=On`.
- All functions which were named `get_gid()` but were returning `hpx::id_type` have been renamed to `get_id()`. To enable the old function names for backwards compatibility, use the cmake configuration option `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY=On`.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #1855[4981] - Completely removing external/endian
- PR #1854[4982] - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1853[4983] - Updating CMake configuration to get correct version of TAU library
- PR #1852[4984] - Fixing Performance Problems with MPI Parcelport
- PR #1851[4985] - Fixing hpx_add_link_flag() and hpx_remove_link_flag()
- PR #1850[4986] - Fixing 1836, adding parallel::sort
- PR #1849[4987] - Fixing configuration for use of more than 64 cores
- PR #1848[4988] - Change default APEX version for release
- PR #1847[4989] - Fix client_base::then on release
- PR #1846[4990] - Removing broken lcos::local::channel from release
- PR #1845[4991] - Adding example demonstrating a possible safe-object implementation to release
- PR #1844[4992] - Removing stubs from accumulator examples

[4980] http://thread.gmane.org/gmane.comp.lib.hpx.devel/196
[4981] https://github.com/STEllAR-GROUP/hpx/pull/1855
[4982] https://github.com/STEllAR-GROUP/hpx/pull/1854
[4983] https://github.com/STEllAR-GROUP/hpx/pull/1853
[4984] https://github.com/STEllAR-GROUP/hpx/pull/1852
[4985] https://github.com/STEllAR-GROUP/hpx/pull/1851
[4986] https://github.com/STEllAR-GROUP/hpx/pull/1850
[4987] https://github.com/STEllAR-GROUP/hpx/pull/1849
[4988] https://github.com/STEllAR-GROUP/hpx/pull/1848
[4989] https://github.com/STEllAR-GROUP/hpx/pull/1847
[4990] https://github.com/STEllAR-GROUP/hpx/pull/1846
[4991] https://github.com/STEllAR-GROUP/hpx/pull/1845
[4992] https://github.com/STEllAR-GROUP/hpx/pull/1844

- PR #1843[4993] - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1841[4994] - Fixing client_base<>::then
- PR #1840[4995] - Adding example demonstrating a possible safe-object implementation
- PR #1838[4996] - Update version rc1
- PR #1837[4997] - Removing broken lcos::local::channel
- PR #1835[4998] - Adding explicit move constructor and assignment operator to hpx::lcos::promise
- PR #1834[4999] - Making hpx::lcos::promise move-only
- PR #1833[5000] - Adding fedora docs
- Issue #1832[5001] - hpx::lcos::promise<> must be move-only
- PR #1831[5002] - Fixing resource manager gcc5.2
- PR #1830[5003] - Fix intel13
- PR #1829[5004] - Unbreaking thread test
- PR #1828[5005] - Fixing #1620
- PR #1827[5006] - Fixing a memory management issue for the Parquet application
- Issue #1826[5007] - Memory management issue in hpx::lcos::promise
- PR #1825[5008] - Adding hpx::components::component and hpx::components::component_base
- PR #1823[5009] - Adding git commit id to circleci build
- PR #1822[5010] - applying fixes suggested by clang 3.7
- PR #1821[5011] - Hyperlink fixes
- PR #1820[5012] - added parallel multi-locality sanity test
- PR #1819[5013] - Fixing #1667
- Issue #1817[5014] - Hyperlinks generated by inspect tool are wrong
- PR #1816[5015] - Support hpxrx

[4993] https://github.com/STEllAR-GROUP/hpx/pull/1843
[4994] https://github.com/STEllAR-GROUP/hpx/pull/1841
[4995] https://github.com/STEllAR-GROUP/hpx/pull/1840
[4996] https://github.com/STEllAR-GROUP/hpx/pull/1838
[4997] https://github.com/STEllAR-GROUP/hpx/pull/1837
[4998] https://github.com/STEllAR-GROUP/hpx/pull/1835
[4999] https://github.com/STEllAR-GROUP/hpx/pull/1834
[5000] https://github.com/STEllAR-GROUP/hpx/pull/1833
[5001] https://github.com/STEllAR-GROUP/hpx/issues/1832
[5002] https://github.com/STEllAR-GROUP/hpx/pull/1831
[5003] https://github.com/STEllAR-GROUP/hpx/pull/1830
[5004] https://github.com/STEllAR-GROUP/hpx/pull/1829
[5005] https://github.com/STEllAR-GROUP/hpx/pull/1828
[5006] https://github.com/STEllAR-GROUP/hpx/pull/1827
[5007] https://github.com/STEllAR-GROUP/hpx/issues/1826
[5008] https://github.com/STEllAR-GROUP/hpx/pull/1825
[5009] https://github.com/STEllAR-GROUP/hpx/pull/1823
[5010] https://github.com/STEllAR-GROUP/hpx/pull/1822
[5011] https://github.com/STEllAR-GROUP/hpx/pull/1821
[5012] https://github.com/STEllAR-GROUP/hpx/pull/1820
[5013] https://github.com/STEllAR-GROUP/hpx/pull/1819
[5014] https://github.com/STEllAR-GROUP/hpx/issues/1817
[5015] https://github.com/STEllAR-GROUP/hpx/pull/1816

- PR #1814[5016] - Fix async to dispatch to the correct locality in all cases
- Issue #1813[5017] - async(launch::. . . , action(), . . .) always invokes locally
- PR #1812[5018] - fixed syntax error in CMakeLists.txt
- PR #1811[5019] - Agas optimizations
- PR #1810[5020] - drop superfluous typedefs
- PR #1809[5021] - Allow HPX to be used as an optional package in 3rd party code
- PR #1808[5022] - Fixing #1723
- PR #1807[5023] - Making sure resolve_localities does not hang during normal operation
- Issue #1806[5024] - Spinlock no longer movable and deletes operator '=', breaks MiniGhost
- Issue #1804[5025] - register_with_basename causes hangs
- PR #1801[5026] - Enhanced the inspect tool to take user directly to the problem with hyperlinks
- Issue #1800[5027] - Problems compiling application on smic
- PR #1799[5028] - Fixing cv exceptions
- PR #1798[5029] - Documentation refactoring & updating
- PR #1797[5030] - Updating the activeharmony CMake module
- PR #1795[5031] - Fixing cv
- PR #1794[5032] - Fix connect with hpx::runtime_mode_connect
- PR #1793[5033] - fix a wrong use of HPX_MAX_CPU_COUNT instead of HPX_HAVE_MAX_CPU_COUNT
- PR #1792[5034] - Allow for default constructed parcel instances to be moved
- PR #1791[5035] - Fix connect with hpx::runtime_mode_connect
- Issue #1790[5036] - assertion `action_.get()` failed: HPX(assertion_failure) when running Octotiger with pull request 1786
- PR #1789[5037] - Fixing discover_counter_types API function
- Issue #1788[5038] - connect with hpx::runtime_mode_connect

[5016] https://github.com/STEllAR-GROUP/hpx/pull/1814
[5017] https://github.com/STEllAR-GROUP/hpx/issues/1813
[5018] https://github.com/STEllAR-GROUP/hpx/pull/1812
[5019] https://github.com/STEllAR-GROUP/hpx/pull/1811
[5020] https://github.com/STEllAR-GROUP/hpx/pull/1810
[5021] https://github.com/STEllAR-GROUP/hpx/pull/1809
[5022] https://github.com/STEllAR-GROUP/hpx/pull/1808
[5023] https://github.com/STEllAR-GROUP/hpx/pull/1807
[5024] https://github.com/STEllAR-GROUP/hpx/issues/1806
[5025] https://github.com/STEllAR-GROUP/hpx/issues/1804
[5026] https://github.com/STEllAR-GROUP/hpx/pull/1801
[5027] https://github.com/STEllAR-GROUP/hpx/issues/1800
[5028] https://github.com/STEllAR-GROUP/hpx/pull/1799
[5029] https://github.com/STEllAR-GROUP/hpx/pull/1798
[5030] https://github.com/STEllAR-GROUP/hpx/pull/1797
[5031] https://github.com/STEllAR-GROUP/hpx/pull/1795
[5032] https://github.com/STEllAR-GROUP/hpx/pull/1794
[5033] https://github.com/STEllAR-GROUP/hpx/pull/1793
[5034] https://github.com/STEllAR-GROUP/hpx/pull/1792
[5035] https://github.com/STEllAR-GROUP/hpx/pull/1791
[5036] https://github.com/STEllAR-GROUP/hpx/issues/1790
[5037] https://github.com/STEllAR-GROUP/hpx/pull/1789
[5038] https://github.com/STEllAR-GROUP/hpx/issues/1788

- Issue #1787[5039] - discover_counter_types not working
- PR #1786[5040] - Changing addressing_service to use std::unordered_map instead of std::map
- PR #1785[5041] - Fix is_iterator for container algorithms
- PR #1784[5042] - Adding new command line options:
- PR #1783[5043] - Minor changes for APEX support
- PR #1782[5044] - Drop legacy forwarding action traits
- PR #1781[5045] - Attempt to resolve the race between cv::wait_xxx and cv::notify_all
- PR #1780[5046] - Removing serialize_sequence
- PR #1779[5047] - Fixed #1501: hwloc configuration options are wrong for MIC
- PR #1778[5048] - Removing ability to enable/disable parcel handling
- PR #1777[5049] - Completely removing stackless threads
- PR #1776[5050] - Cleaning up util/plugin
- PR #1775[5051] - Agas fixes
- PR #1774[5052] - Action invocation count
- PR #1773[5053] - replaced MSVC variable with WIN32
- PR #1772[5054] - Fixing Problems in MPI parcelport and future serialization.
- PR #1771[5055] - Fixing intel 13 compiler errors related to variadic template template parameters for `lcos::when_` tests
- PR #1770[5056] - Forwarding decay to `std::`
- PR #1769[5057] - Add more characters with special regex meaning to the existing patch
- PR #1768[5058] - Adding test for receive_buffer
- PR #1767[5059] - Making sure that uptime counter throws exception on any attempt to be reset
- PR #1766[5060] - Cleaning up code related to throttling scheduler
- PR #1765[5061] - Restricting thread_data to creating only with intrusive_pointers

[5039] https://github.com/STEllAR-GROUP/hpx/issues/1787
[5040] https://github.com/STEllAR-GROUP/hpx/pull/1786
[5041] https://github.com/STEllAR-GROUP/hpx/pull/1785
[5042] https://github.com/STEllAR-GROUP/hpx/pull/1784
[5043] https://github.com/STEllAR-GROUP/hpx/pull/1783
[5044] https://github.com/STEllAR-GROUP/hpx/pull/1782
[5045] https://github.com/STEllAR-GROUP/hpx/pull/1781
[5046] https://github.com/STEllAR-GROUP/hpx/pull/1780
[5047] https://github.com/STEllAR-GROUP/hpx/pull/1779
[5048] https://github.com/STEllAR-GROUP/hpx/pull/1778
[5049] https://github.com/STEllAR-GROUP/hpx/pull/1777
[5050] https://github.com/STEllAR-GROUP/hpx/pull/1776
[5051] https://github.com/STEllAR-GROUP/hpx/pull/1775
[5052] https://github.com/STEllAR-GROUP/hpx/pull/1774
[5053] https://github.com/STEllAR-GROUP/hpx/pull/1773
[5054] https://github.com/STEllAR-GROUP/hpx/pull/1772
[5055] https://github.com/STEllAR-GROUP/hpx/pull/1771
[5056] https://github.com/STEllAR-GROUP/hpx/pull/1770
[5057] https://github.com/STEllAR-GROUP/hpx/pull/1769
[5058] https://github.com/STEllAR-GROUP/hpx/pull/1768
[5059] https://github.com/STEllAR-GROUP/hpx/pull/1767
[5060] https://github.com/STEllAR-GROUP/hpx/pull/1766
[5061] https://github.com/STEllAR-GROUP/hpx/pull/1765

- PR #1764[5062] - Fixing 1763
- Issue #1763[5063] - UB in thread_data::operator delete
- PR #1762[5064] - Making sure all serialization registries/factories are unique
- PR #1761[5065] - Fixed #1751: hpx::future::wait_for fails a simple test
- PR #1758[5066] - Fixing #1757
- Issue #1757[5067] - pinning not correct using –hpx:bind
- Issue #1756[5068] - compilation error with MinGW
- PR #1755[5069] - Making output serialization const-correct
- Issue #1753[5070] - HPX performance degrades with time since execution begins
- Issue #1752[5071] - Error in AGAS
- Issue #1751[5072] - hpx::future::wait_for fails a simple test
- PR #1750[5073] - Removing hpx_fwd.hpp includes
- PR #1749[5074] - Simplify result_of and friends
- PR #1747[5075] - Removed superfluous code from message_buffer.hpp
- PR #1746[5076] - Tuple dependencies
- Issue #1745[5077] - Broken when_some which takes iterators
- PR #1744[5078] - Refining archive interface
- PR #1743[5079] - Fixing when_all when only a single future is passed
- PR #1742[5080] - Config includes
- PR #1741[5081] - Os executors
- Issue #1740[5082] - hpx::promise has some problems
- PR #1739[5083] - Parallel composition with generic containers
- Issue #1738[5084] - After building program and successfully linking to a version of hpx DHPX_DIR seems to be ignored

[5062] https://github.com/STEllAR-GROUP/hpx/pull/1764
[5063] https://github.com/STEllAR-GROUP/hpx/issues/1763
[5064] https://github.com/STEllAR-GROUP/hpx/pull/1762
[5065] https://github.com/STEllAR-GROUP/hpx/pull/1761
[5066] https://github.com/STEllAR-GROUP/hpx/pull/1758
[5067] https://github.com/STEllAR-GROUP/hpx/issues/1757
[5068] https://github.com/STEllAR-GROUP/hpx/issues/1756
[5069] https://github.com/STEllAR-GROUP/hpx/pull/1755
[5070] https://github.com/STEllAR-GROUP/hpx/issues/1753
[5071] https://github.com/STEllAR-GROUP/hpx/issues/1752
[5072] https://github.com/STEllAR-GROUP/hpx/issues/1751
[5073] https://github.com/STEllAR-GROUP/hpx/pull/1750
[5074] https://github.com/STEllAR-GROUP/hpx/pull/1749
[5075] https://github.com/STEllAR-GROUP/hpx/pull/1747
[5076] https://github.com/STEllAR-GROUP/hpx/pull/1746
[5077] https://github.com/STEllAR-GROUP/hpx/issues/1745
[5078] https://github.com/STEllAR-GROUP/hpx/pull/1744
[5079] https://github.com/STEllAR-GROUP/hpx/pull/1743
[5080] https://github.com/STEllAR-GROUP/hpx/pull/1742
[5081] https://github.com/STEllAR-GROUP/hpx/pull/1741
[5082] https://github.com/STEllAR-GROUP/hpx/issues/1740
[5083] https://github.com/STEllAR-GROUP/hpx/pull/1739
[5084] https://github.com/STEllAR-GROUP/hpx/issues/1738

- Issue #1737[5085] - Uptime problems
- PR #1736[5086] - added convenience c-tor and begin()/end() to serialize_buffer
- PR #1735[5087] - Config includes
- PR #1734[5088] - Fixed #1688: Add timer counters for tfunc_total and exec_total
- Issue #1733[5089] - Add unit test for hpx/lcos/local/receive_buffer.hpp
- PR #1732[5090] - Renaming get_os_thread_count
- PR #1731[5091] - Basename registration
- Issue #1730[5092] - Use after move of thread_init_data
- PR #1729[5093] - Rewriting channel based on new gate component
- PR #1728[5094] - Fixing #1722
- PR #1727[5095] - Fixing compile problems with apply_colocated
- PR #1726[5096] - Apex integration
- PR #1725[5097] - fixed test timeouts
- PR #1724[5098] - Renaming vector
- Issue #1723[5099] - Drop support for intel compilers and gcc 4.4. based standard libs
- Issue #1722[5100] - Add support for detecting non-ready futures before serialization
- PR #1721[5101] - Unifying parallel executors, initializing from launch policy
- PR #1720[5102] - dropped superfluous typedef
- Issue #1718[5103] - Windows 10 x64, VS 2015 - Unknown CMake command "add_hpx_pseudo_target".
- PR #1717[5104] - Timed executor traits for thread-executors
- PR #1716[5105] - serialization of arrays didn't work with non-pod types. fixed
- PR #1715[5106] - List serialization
- PR #1714[5107] - changing misspellings

5085 https://github.com/STEllAR-GROUP/hpx/issues/1737
5086 https://github.com/STEllAR-GROUP/hpx/pull/1736
5087 https://github.com/STEllAR-GROUP/hpx/pull/1735
5088 https://github.com/STEllAR-GROUP/hpx/pull/1734
5089 https://github.com/STEllAR-GROUP/hpx/issues/1733
5090 https://github.com/STEllAR-GROUP/hpx/pull/1732
5091 https://github.com/STEllAR-GROUP/hpx/pull/1731
5092 https://github.com/STEllAR-GROUP/hpx/issues/1730
5093 https://github.com/STEllAR-GROUP/hpx/pull/1729
5094 https://github.com/STEllAR-GROUP/hpx/pull/1728
5095 https://github.com/STEllAR-GROUP/hpx/pull/1727
5096 https://github.com/STEllAR-GROUP/hpx/pull/1726
5097 https://github.com/STEllAR-GROUP/hpx/pull/1725
5098 https://github.com/STEllAR-GROUP/hpx/pull/1724
5099 https://github.com/STEllAR-GROUP/hpx/issues/1723
5100 https://github.com/STEllAR-GROUP/hpx/issues/1722
5101 https://github.com/STEllAR-GROUP/hpx/pull/1721
5102 https://github.com/STEllAR-GROUP/hpx/pull/1720
5103 https://github.com/STEllAR-GROUP/hpx/issues/1718
5104 https://github.com/STEllAR-GROUP/hpx/pull/1717
5105 https://github.com/STEllAR-GROUP/hpx/pull/1716
5106 https://github.com/STEllAR-GROUP/hpx/pull/1715
5107 https://github.com/STEllAR-GROUP/hpx/pull/1714

- PR #1713[5108] - Fixed distribution policy executors
- PR #1712[5109] - Moving library detection to be executed after feature tests
- PR #1711[5110] - Simplify parcel
- PR #1710[5111] - Compile only tests
- PR #1709[5112] - Implemented timed executors
- PR #1708[5113] - Implement parallel::executor_traits for thread-executors
- PR #1707[5114] - Various fixes to threads::executors to make custom schedulers work
- PR #1706[5115] - Command line option –hpx:cores does not work as expected
- Issue #1705[5116] - command line option –hpx:cores does not work as expected
- PR #1704[5117] - vector deserialization is speeded up a little
- PR #1703[5118] - Fixing shared_mutes
- Issue #1702[5119] - Shared_mutex does not compile with no_mutex cond_var
- PR #1701[5120] - Add distribution_policy_executor
- PR #1700[5121] - Executor parameters
- PR #1699[5122] - Readers writer lock
- PR #1698[5123] - Remove leftovers
- PR #1697[5124] - Fixing held locks
- PR #1696[5125] - Modified Scan Partitioner for Algorithms
- PR #1695[5126] - This thread executors
- PR #1694[5127] - Fixed #1688: Add timer counters for tfunc_total and exec_total
- PR #1693[5128] - Fix #1691: is_executor template specification fails for inherited executors
- PR #1692[5129] - Fixed #1662: Possible exception source in coalescing_message_handler
- Issue #1691[5130] - is_executor template specification fails for inherited executors

5108 https://github.com/STEllAR-GROUP/hpx/pull/1713
5109 https://github.com/STEllAR-GROUP/hpx/pull/1712
5110 https://github.com/STEllAR-GROUP/hpx/pull/1711
5111 https://github.com/STEllAR-GROUP/hpx/pull/1710
5112 https://github.com/STEllAR-GROUP/hpx/pull/1709
5113 https://github.com/STEllAR-GROUP/hpx/pull/1708
5114 https://github.com/STEllAR-GROUP/hpx/pull/1707
5115 https://github.com/STEllAR-GROUP/hpx/pull/1706
5116 https://github.com/STEllAR-GROUP/hpx/issues/1705
5117 https://github.com/STEllAR-GROUP/hpx/pull/1704
5118 https://github.com/STEllAR-GROUP/hpx/pull/1703
5119 https://github.com/STEllAR-GROUP/hpx/issues/1702
5120 https://github.com/STEllAR-GROUP/hpx/pull/1701
5121 https://github.com/STEllAR-GROUP/hpx/pull/1700
5122 https://github.com/STEllAR-GROUP/hpx/pull/1699
5123 https://github.com/STEllAR-GROUP/hpx/pull/1698
5124 https://github.com/STEllAR-GROUP/hpx/pull/1697
5125 https://github.com/STEllAR-GROUP/hpx/pull/1696
5126 https://github.com/STEllAR-GROUP/hpx/pull/1695
5127 https://github.com/STEllAR-GROUP/hpx/pull/1694
5128 https://github.com/STEllAR-GROUP/hpx/pull/1693
5129 https://github.com/STEllAR-GROUP/hpx/pull/1692
5130 https://github.com/STEllAR-GROUP/hpx/issues/1691

- PR #1690[5131] - added macro for non-intrusive serialization of classes without a default c-tor
- PR #1689[5132] - Replace value_or_error with custom storage, unify future_data state
- Issue #1688[5133] - Add timer counters for tfunc_total and exec_total
- PR #1687[5134] - Fixed interval timer
- PR #1686[5135] - Fixing cmake warnings about not existing pseudo target dependencies
- PR #1685[5136] - Converting partitioners to use bulk async execute
- PR #1683[5137] - Adds a tool for inspect that checks for character limits
- PR #1682[5138] - Change project name to (uppercase) HPX
- PR #1681[5139] - Counter shortnames
- PR #1680[5140] - Extended Non-intrusive Serialization to Ease Usage for Library Developers
- PR #1679[5141] - Working on 1544: More executor changes
- PR #1678[5142] - Transpose fixes
- PR #1677[5143] - Improve Boost compatibility check
- PR #1676[5144] - 1d stencil fix
- Issue #1675[5145] - hpx project name is not HPX
- PR #1674[5146] - Fixing the MPI parcelport
- PR #1673[5147] - added move semantics to map/vector deserialization
- PR #1672[5148] - Vs2015 await
- PR #1671[5149] - Adapt transform for #1668
- PR #1670[5150] - Started to work on #1668
- PR #1669[5151] - Add this_thread_executors
- Issue #1667[5152] - Apple build instructions in docs are out of date
- PR #1666[5153] - Apex integration

5131 https://github.com/STEllAR-GROUP/hpx/pull/1690
5132 https://github.com/STEllAR-GROUP/hpx/pull/1689
5133 https://github.com/STEllAR-GROUP/hpx/issues/1688
5134 https://github.com/STEllAR-GROUP/hpx/pull/1687
5135 https://github.com/STEllAR-GROUP/hpx/pull/1686
5136 https://github.com/STEllAR-GROUP/hpx/pull/1685
5137 https://github.com/STEllAR-GROUP/hpx/pull/1683
5138 https://github.com/STEllAR-GROUP/hpx/pull/1682
5139 https://github.com/STEllAR-GROUP/hpx/pull/1681
5140 https://github.com/STEllAR-GROUP/hpx/pull/1680
5141 https://github.com/STEllAR-GROUP/hpx/pull/1679
5142 https://github.com/STEllAR-GROUP/hpx/pull/1678
5143 https://github.com/STEllAR-GROUP/hpx/pull/1677
5144 https://github.com/STEllAR-GROUP/hpx/pull/1676
5145 https://github.com/STEllAR-GROUP/hpx/issues/1675
5146 https://github.com/STEllAR-GROUP/hpx/pull/1674
5147 https://github.com/STEllAR-GROUP/hpx/pull/1673
5148 https://github.com/STEllAR-GROUP/hpx/pull/1672
5149 https://github.com/STEllAR-GROUP/hpx/pull/1671
5150 https://github.com/STEllAR-GROUP/hpx/pull/1670
5151 https://github.com/STEllAR-GROUP/hpx/pull/1669
5152 https://github.com/STEllAR-GROUP/hpx/issues/1667
5153 https://github.com/STEllAR-GROUP/hpx/pull/1666

- PR #1665[5154] - Fixes an error with the whitespace check that showed the incorrect location of the error
- Issue #1664[5155] - Inspect tool found incorrect endline whitespace
- PR #1663[5156] - Improve use of locks
- Issue #1662[5157] - Possible exception source in coalescing_message_handler
- PR #1661[5158] - Added support for 128bit number serialization
- PR #1660[5159] - Serialization 128bits
- PR #1659[5160] - Implemented inner_product and adjacent_diff algos
- PR #1658[5161] - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1657[5162] - Use of shared_ptr in io_service_pool changed to unique_ptr
- Issue #1656[5163] - 1d_stencil codes all have wrong factor
- PR #1654[5164] - When using runtime_mode_connect, find the correct localhost public ip address
- PR #1653[5165] - Fixing 1617
- PR #1652[5166] - Remove traits::action_may_require_id_splitting
- PR #1651[5167] - Fixed performance counters related to AGAS cache timings
- PR #1650[5168] - Remove leftovers of traits::type_size
- PR #1649[5169] - Shorten target names on Windows to shorten used path names
- PR #1648[5170] - Fixing problems introduced by merging #1623 for older compilers
- PR #1647[5171] - Simplify running automatic builds on Windows
- Issue #1646[5172] - Cache insert and update performance counters are broken
- Issue #1644[5173] - Remove leftovers of traits::type_size
- Issue #1643[5174] - Remove traits::action_may_require_id_splitting
- PR #1642[5175] - Adds spell checker to the inspect tool for qbk and doxygen comments
- PR #1640[5176] - First step towards fixing 688

5154 https://github.com/STEllAR-GROUP/hpx/pull/1665
5155 https://github.com/STEllAR-GROUP/hpx/issues/1664
5156 https://github.com/STEllAR-GROUP/hpx/pull/1663
5157 https://github.com/STEllAR-GROUP/hpx/issues/1662
5158 https://github.com/STEllAR-GROUP/hpx/pull/1661
5159 https://github.com/STEllAR-GROUP/hpx/pull/1660
5160 https://github.com/STEllAR-GROUP/hpx/pull/1659
5161 https://github.com/STEllAR-GROUP/hpx/pull/1658
5162 https://github.com/STEllAR-GROUP/hpx/pull/1657
5163 https://github.com/STEllAR-GROUP/hpx/issues/1656
5164 https://github.com/STEllAR-GROUP/hpx/pull/1654
5165 https://github.com/STEllAR-GROUP/hpx/pull/1653
5166 https://github.com/STEllAR-GROUP/hpx/pull/1652
5167 https://github.com/STEllAR-GROUP/hpx/pull/1651
5168 https://github.com/STEllAR-GROUP/hpx/pull/1650
5169 https://github.com/STEllAR-GROUP/hpx/pull/1649
5170 https://github.com/STEllAR-GROUP/hpx/pull/1648
5171 https://github.com/STEllAR-GROUP/hpx/pull/1647
5172 https://github.com/STEllAR-GROUP/hpx/issues/1646
5173 https://github.com/STEllAR-GROUP/hpx/issues/1644
5174 https://github.com/STEllAR-GROUP/hpx/issues/1643
5175 https://github.com/STEllAR-GROUP/hpx/pull/1642
5176 https://github.com/STEllAR-GROUP/hpx/pull/1640

- PR #1639[5177] - Re-apply remaining changes from limit_dataflow_recursion branch
- PR #1638[5178] - This fixes possible deadlock in the test ignore_while_locked_1485
- PR #1637[5179] - Fixing hpx::wait_all() invoked with two vector<future<T>>
- PR #1636[5180] - Partially re-apply changes from limit_dataflow_recursion branch
- PR #1635[5181] - Adding missing test for #1572
- PR #1634[5182] - Revert "Limit recursion-depth in dataflow to a configurable constant"
- PR #1633[5183] - Add command line option to ignore batch environment
- PR #1631[5184] - hpx::lcos::queue exhibits strange behavior
- PR #1630[5185] - Fixed endline_whitespace_check.cpp to detect lines with only whitespace
- Issue #1629[5186] - Inspect trailing whitespace checker problem
- PR #1628[5187] - Removed meaningless const qualifiers. Minor icpc fix.
- PR #1627[5188] - Fixing the queue LCO and add example demonstrating its use
- PR #1626[5189] - Deprecating get_gid(), add get_id() and get_unmanaged_id()
- PR #1625[5190] - Allowing to specify whether to send credits along with message
- Issue #1624[5191] - Lifetime issue
- Issue #1623[5192] - hpx::wait_all() invoked with two vector<future<T>> fails
- PR #1622[5193] - Executor partitioners
- PR #1621[5194] - Clean up coroutines implementation
- Issue #1620[5195] - Revert #1535
- PR #1619[5196] - Fix result type calculation for hpx::make_continuation
- PR #1618[5197] - Fixing RDTSC on Xeon/Phi
- Issue #1617[5198] - hpx cmake not working when run as a subproject
- Issue #1616[5199] - cmake problem resulting in RDTSC not working correctly for Xeon Phi creates very strange results for duration counters

5177 https://github.com/STEllAR-GROUP/hpx/pull/1639
5178 https://github.com/STEllAR-GROUP/hpx/pull/1638
5179 https://github.com/STEllAR-GROUP/hpx/pull/1637
5180 https://github.com/STEllAR-GROUP/hpx/pull/1636
5181 https://github.com/STEllAR-GROUP/hpx/pull/1635
5182 https://github.com/STEllAR-GROUP/hpx/pull/1634
5183 https://github.com/STEllAR-GROUP/hpx/pull/1633
5184 https://github.com/STEllAR-GROUP/hpx/pull/1631
5185 https://github.com/STEllAR-GROUP/hpx/pull/1630
5186 https://github.com/STEllAR-GROUP/hpx/issues/1629
5187 https://github.com/STEllAR-GROUP/hpx/pull/1628
5188 https://github.com/STEllAR-GROUP/hpx/pull/1627
5189 https://github.com/STEllAR-GROUP/hpx/pull/1626
5190 https://github.com/STEllAR-GROUP/hpx/pull/1625
5191 https://github.com/STEllAR-GROUP/hpx/issues/1624
5192 https://github.com/STEllAR-GROUP/hpx/issues/1623
5193 https://github.com/STEllAR-GROUP/hpx/pull/1622
5194 https://github.com/STEllAR-GROUP/hpx/pull/1621
5195 https://github.com/STEllAR-GROUP/hpx/issues/1620
5196 https://github.com/STEllAR-GROUP/hpx/pull/1619
5197 https://github.com/STEllAR-GROUP/hpx/pull/1618
5198 https://github.com/STEllAR-GROUP/hpx/issues/1617
5199 https://github.com/STEllAR-GROUP/hpx/issues/1616

- Issue #1615[5200] - hpx::make_continuation requires input and output to be the same
- PR #1614[5201] - Fixed remove copy test
- Issue #1613[5202] - Dataflow causes stack overflow
- PR #1612[5203] - Modified foreach partitioner to use bulk execute
- PR #1611[5204] - Limit recursion-depth in dataflow to a configurable constant
- PR #1610[5205] - Increase timeout for CircleCI
- PR #1609[5206] - Refactoring thread manager, mainly extracting thread pool
- PR #1608[5207] - Fixed running multiple localities without localities parameter
- PR #1607[5208] - More algorithm fixes to adjacentfind
- Issue #1606[5209] - Running without localities parameter binds to bogus port range
- Issue #1605[5210] - Too many serializations
- PR #1604[5211] - Changes the HPX image into a hyperlink
- PR #1601[5212] - Fixing problems with remove_copy algorithm tests
- PR #1600[5213] - Actions with ids cleanup
- PR #1599[5214] - Duplicate binding of global ids should fail
- PR #1598[5215] - Fixing array access
- PR #1597[5216] - Improved the reliability of connecting/disconnecting localities
- Issue #1596[5217] - Duplicate id binding should fail
- PR #1595[5218] - Fixing more cmake config constants
- PR #1594[5219] - Fixing preprocessor constant used to enable C++11 chrono
- PR #1593[5220] - Adding operator|() for hpx::launch
- Issue #1592[5221] - Error (typo) in the docs
- Issue #1590[5222] - CMake fails when CMAKE_BINARY_DIR contains '+'.

5200 https://github.com/STEllAR-GROUP/hpx/issues/1615
5201 https://github.com/STEllAR-GROUP/hpx/pull/1614
5202 https://github.com/STEllAR-GROUP/hpx/issues/1613
5203 https://github.com/STEllAR-GROUP/hpx/pull/1612
5204 https://github.com/STEllAR-GROUP/hpx/pull/1611
5205 https://github.com/STEllAR-GROUP/hpx/pull/1610
5206 https://github.com/STEllAR-GROUP/hpx/pull/1609
5207 https://github.com/STEllAR-GROUP/hpx/pull/1608
5208 https://github.com/STEllAR-GROUP/hpx/pull/1607
5209 https://github.com/STEllAR-GROUP/hpx/issues/1606
5210 https://github.com/STEllAR-GROUP/hpx/issues/1605
5211 https://github.com/STEllAR-GROUP/hpx/pull/1604
5212 https://github.com/STEllAR-GROUP/hpx/pull/1601
5213 https://github.com/STEllAR-GROUP/hpx/pull/1600
5214 https://github.com/STEllAR-GROUP/hpx/pull/1599
5215 https://github.com/STEllAR-GROUP/hpx/pull/1598
5216 https://github.com/STEllAR-GROUP/hpx/pull/1597
5217 https://github.com/STEllAR-GROUP/hpx/issues/1596
5218 https://github.com/STEllAR-GROUP/hpx/pull/1595
5219 https://github.com/STEllAR-GROUP/hpx/pull/1594
5220 https://github.com/STEllAR-GROUP/hpx/pull/1593
5221 https://github.com/STEllAR-GROUP/hpx/issues/1592
5222 https://github.com/STEllAR-GROUP/hpx/issues/1590

- Issue #1589[5223] - Disconnecting a locality results in segfault using heartbeat example
- PR #1588[5224] - Fix doc string for config option HPX_WITH_EXAMPLES
- PR #1586[5225] - Fixing 1493
- PR #1585[5226] - Additional Check for Inspect Tool to detect Endline Whitespace
- Issue #1584[5227] - Clean up coroutines implementation
- PR #1583[5228] - Adding a check for end line whitespace
- PR #1582[5229] - Attempt to fix assert firing after scheduling loop was exited
- PR #1581[5230] - Fixed adjacentfind_binary test
- PR #1580[5231] - Prevent some of the internal cmake lists from growing indefinitely
- PR #1579[5232] - Removing type_size trait, replacing it with special archive type
- Issue #1578[5233] - Remove demangle_helper
- PR #1577[5234] - Get ptr problems
- Issue #1576[5235] - Refactor async, dataflow, and future::then
- PR #1575[5236] - Fixing tests for parallel rotate
- PR #1574[5237] - Cleaning up schedulers
- PR #1573[5238] - Fixing thread pool executor
- PR #1572[5239] - Fixing number of configured localities
- PR #1571[5240] - Reimplement decay
- PR #1570[5241] - Refactoring async, apply, and dataflow APIs
- PR #1569[5242] - Changed range for mach-o library lookup
- PR #1568[5243] - Mark decltype support as required
- PR #1567[5244] - Removed const from algorithms
- Issue #1566[5245] - CMAKE Configuration Test Failures for clang 3.5 on debian

5223 https://github.com/STEllAR-GROUP/hpx/issues/1589
5224 https://github.com/STEllAR-GROUP/hpx/pull/1588
5225 https://github.com/STEllAR-GROUP/hpx/pull/1586
5226 https://github.com/STEllAR-GROUP/hpx/pull/1585
5227 https://github.com/STEllAR-GROUP/hpx/issues/1584
5228 https://github.com/STEllAR-GROUP/hpx/pull/1583
5229 https://github.com/STEllAR-GROUP/hpx/pull/1582
5230 https://github.com/STEllAR-GROUP/hpx/pull/1581
5231 https://github.com/STEllAR-GROUP/hpx/pull/1580
5232 https://github.com/STEllAR-GROUP/hpx/pull/1579
5233 https://github.com/STEllAR-GROUP/hpx/issues/1578
5234 https://github.com/STEllAR-GROUP/hpx/pull/1577
5235 https://github.com/STEllAR-GROUP/hpx/issues/1576
5236 https://github.com/STEllAR-GROUP/hpx/pull/1575
5237 https://github.com/STEllAR-GROUP/hpx/pull/1574
5238 https://github.com/STEllAR-GROUP/hpx/pull/1573
5239 https://github.com/STEllAR-GROUP/hpx/pull/1572
5240 https://github.com/STEllAR-GROUP/hpx/pull/1571
5241 https://github.com/STEllAR-GROUP/hpx/pull/1570
5242 https://github.com/STEllAR-GROUP/hpx/pull/1569
5243 https://github.com/STEllAR-GROUP/hpx/pull/1568
5244 https://github.com/STEllAR-GROUP/hpx/pull/1567
5245 https://github.com/STEllAR-GROUP/hpx/issues/1566

- PR #1565[5246] - Dylib support
- PR #1564[5247] - Converted partitioners and some algorithms to use executors
- PR #1563[5248] - Fix several #includes for Boost.Preprocessor
- PR #1562[5249] - Adding configuration option disabling/enabling all message handlers
- PR #1561[5250] - Removed all occurrences of boost::move replacing it with std::move
- Issue #1560[5251] - Leftover HPX_REGISTER_ACTION_DECLARATION_2
- PR #1558[5252] - Revisit async/apply SFINAE conditions
- PR #1557[5253] - Removing type_size trait, replacing it with special archive type
- PR #1556[5254] - Executor algorithms
- PR #1555[5255] - Remove the necessity to specify archive flags on the receiving end
- PR #1554[5256] - Removing obsolete Boost.Serialization macros
- PR #1553[5257] - Properly fix HPX_DEFINE_*_ACTION macros
- PR #1552[5258] - Fixed algorithms relying on copy_if implementation
- PR #1551[5259] - Pxfs - Modifying FindOrangeFS.cmake based on OrangeFS 2.9.X
- Issue #1550[5260] - Passing plain identifier inside HPX_DEFINE_PLAIN_ACTION_1
- PR #1549[5261] - Fixing intel14/libstdc++4.4
- PR #1548[5262] - Moving raw_ptr to detail namespace
- PR #1547[5263] - Adding support for executors to future.then
- PR #1546[5264] - Executor traits result types
- PR #1545[5265] - Integrate executors with dataflow
- PR #1543[5266] - Fix potential zero-copy for primarynamespace::bulk_service_async et.al.
- PR #1542[5267] - Merging HPX0.9.10 into pxfs branch
- PR #1541[5268] - Removed stale cmake tests, unused since the great cmake refactoring

5246 https://github.com/STEllAR-GROUP/hpx/pull/1565
5247 https://github.com/STEllAR-GROUP/hpx/pull/1564
5248 https://github.com/STEllAR-GROUP/hpx/pull/1563
5249 https://github.com/STEllAR-GROUP/hpx/pull/1562
5250 https://github.com/STEllAR-GROUP/hpx/pull/1561
5251 https://github.com/STEllAR-GROUP/hpx/issues/1560
5252 https://github.com/STEllAR-GROUP/hpx/pull/1558
5253 https://github.com/STEllAR-GROUP/hpx/pull/1557
5254 https://github.com/STEllAR-GROUP/hpx/pull/1556
5255 https://github.com/STEllAR-GROUP/hpx/pull/1555
5256 https://github.com/STEllAR-GROUP/hpx/pull/1554
5257 https://github.com/STEllAR-GROUP/hpx/pull/1553
5258 https://github.com/STEllAR-GROUP/hpx/pull/1552
5259 https://github.com/STEllAR-GROUP/hpx/pull/1551
5260 https://github.com/STEllAR-GROUP/hpx/issues/1550
5261 https://github.com/STEllAR-GROUP/hpx/pull/1549
5262 https://github.com/STEllAR-GROUP/hpx/pull/1548
5263 https://github.com/STEllAR-GROUP/hpx/pull/1547
5264 https://github.com/STEllAR-GROUP/hpx/pull/1546
5265 https://github.com/STEllAR-GROUP/hpx/pull/1545
5266 https://github.com/STEllAR-GROUP/hpx/pull/1543
5267 https://github.com/STEllAR-GROUP/hpx/pull/1542
5268 https://github.com/STEllAR-GROUP/hpx/pull/1541

- PR #1540[5269] - Fix idle-rate on platforms without TSC
- PR #1539[5270] - Reporting situation if zero-copy-serialization was performed by a parcel generated from a plain apply/async
- PR #1538[5271] - Changed return type of bulk executors and added test
- Issue #1537[5272] - Incorrect cpuid config tests
- PR #1536[5273] - Changed return type of bulk executors and added test
- PR #1535[5274] - Make sure promise::get_gid() can be called more than once
- PR #1534[5275] - Fixed async_callback with bound callback
- PR #1533[5276] - Updated the link in the documentation to a publically- accessible URL
- PR #1532[5277] - Make sure sync primitives are not copyable nor movable
- PR #1531[5278] - Fix unwrapped issue with future ranges of void type
- PR #1530[5279] - Serialization complex
- Issue #1528[5280] - Unwrapped issue with future<void>
- Issue #1527[5281] - HPX does not build with Boost 1.58.0
- PR #1526[5282] - Added support for boost.multi_array serialization
- PR #1525[5283] - Properly handle deferred futures, fixes #1506
- PR #1524[5284] - Making sure invalid action argument types generate clear error message
- Issue #1522[5285] - Need serialization support for boost multi array
- Issue #1521[5286] - Remote async and zero-copy serialization optimizations don't play well together
- PR #1520[5287] - Fixing UB whil registering polymorphic classes for serialization
- PR #1519[5288] - Making detail::condition_variable safe to use
- PR #1518[5289] - Fix when_some bug missing indices in its result
- Issue #1517[5290] - Typo may affect CMake build system tests
- PR #1516[5291] - Fixing Posix context

5269 https://github.com/STEllAR-GROUP/hpx/pull/1540
5270 https://github.com/STEllAR-GROUP/hpx/pull/1539
5271 https://github.com/STEllAR-GROUP/hpx/pull/1538
5272 https://github.com/STEllAR-GROUP/hpx/issues/1537
5273 https://github.com/STEllAR-GROUP/hpx/pull/1536
5274 https://github.com/STEllAR-GROUP/hpx/pull/1535
5275 https://github.com/STEllAR-GROUP/hpx/pull/1534
5276 https://github.com/STEllAR-GROUP/hpx/pull/1533
5277 https://github.com/STEllAR-GROUP/hpx/pull/1532
5278 https://github.com/STEllAR-GROUP/hpx/pull/1531
5279 https://github.com/STEllAR-GROUP/hpx/pull/1530
5280 https://github.com/STEllAR-GROUP/hpx/issues/1528
5281 https://github.com/STEllAR-GROUP/hpx/issues/1527
5282 https://github.com/STEllAR-GROUP/hpx/pull/1526
5283 https://github.com/STEllAR-GROUP/hpx/pull/1525
5284 https://github.com/STEllAR-GROUP/hpx/pull/1524
5285 https://github.com/STEllAR-GROUP/hpx/issues/1522
5286 https://github.com/STEllAR-GROUP/hpx/issues/1521
5287 https://github.com/STEllAR-GROUP/hpx/pull/1520
5288 https://github.com/STEllAR-GROUP/hpx/pull/1519
5289 https://github.com/STEllAR-GROUP/hpx/pull/1518
5290 https://github.com/STEllAR-GROUP/hpx/issues/1517
5291 https://github.com/STEllAR-GROUP/hpx/pull/1516

- PR #1515[5292] - Fixing Posix context
- PR #1514[5293] - Correct problems with loading dynamic components
- PR #1513[5294] - Fixing intel glibc4 4
- Issue #1508[5295] - memory and papi counters do not work
- Issue #1507[5296] - Unrecognized Command Line Option Error causing exit status 0
- Issue #1506[5297] - Properly handle deferred futures
- PR #1505[5298] - Adding #include - would not compile without this
- Issue #1502[5299] - `boost::filesystem::exists` throws unexpected exception
- Issue #1501[5300] - hwloc configuration options are wrong for MIC
- PR #1504[5301] - Making sure boost::filesystem::exists() does not throw
- PR #1500[5302] - Exit application on `--hpx:version`/`-v` and `--hpx:info`
- PR #1498[5303] - Extended task block
- PR #1497[5304] - Unique ptr serialization
- PR #1496[5305] - Unique ptr serialization (closed)
- PR #1495[5306] - Switching circleci build type to debug
- Issue #1494[5307] - `--hpx:version`/`-v` does not exit after printing version information
- Issue #1493[5308] - add an `hpx_` prefix to libraries and components to avoid name conflicts
- Issue #1492[5309] - Define and ensure limitations for arguments to async/apply
- PR #1489[5310] - Enable idle rate counter on demand
- PR #1488[5311] - Made sure `detail::condition_variable` can be safely destroyed
- PR #1487[5312] - Introduced default (main) template implementation for `ignore_while_checking`
- PR #1486[5313] - Add HPX inspect tool
- Issue #1485[5314] - `ignore_while_locked` doesn't support all Lockable types

5292 https://github.com/STEllAR-GROUP/hpx/pull/1515
5293 https://github.com/STEllAR-GROUP/hpx/pull/1514
5294 https://github.com/STEllAR-GROUP/hpx/pull/1513
5295 https://github.com/STEllAR-GROUP/hpx/issues/1508
5296 https://github.com/STEllAR-GROUP/hpx/issues/1507
5297 https://github.com/STEllAR-GROUP/hpx/issues/1506
5298 https://github.com/STEllAR-GROUP/hpx/pull/1505
5299 https://github.com/STEllAR-GROUP/hpx/issues/1502
5300 https://github.com/STEllAR-GROUP/hpx/issues/1501
5301 https://github.com/STEllAR-GROUP/hpx/pull/1504
5302 https://github.com/STEllAR-GROUP/hpx/pull/1500
5303 https://github.com/STEllAR-GROUP/hpx/pull/1498
5304 https://github.com/STEllAR-GROUP/hpx/pull/1497
5305 https://github.com/STEllAR-GROUP/hpx/pull/1496
5306 https://github.com/STEllAR-GROUP/hpx/pull/1495
5307 https://github.com/STEllAR-GROUP/hpx/issues/1494
5308 https://github.com/STEllAR-GROUP/hpx/issues/1493
5309 https://github.com/STEllAR-GROUP/hpx/issues/1492
5310 https://github.com/STEllAR-GROUP/hpx/pull/1489
5311 https://github.com/STEllAR-GROUP/hpx/pull/1488
5312 https://github.com/STEllAR-GROUP/hpx/pull/1487
5313 https://github.com/STEllAR-GROUP/hpx/pull/1486
5314 https://github.com/STEllAR-GROUP/hpx/issues/1485

- PR #1484[5315] - Docker image generation
- PR #1483[5316] - Move external endian library into HPX
- PR #1482[5317] - Actions with integer type ids
- Issue #1481[5318] - Sync primitives safe destruction
- Issue #1480[5319] - Move external/boost/endian into hpx/util
- Issue #1478[5320] - Boost inspect violations
- PR #1479[5321] - Adds serialization for arrays; some further/minor fixes
- PR #1477[5322] - Fixing problems with the Intel compiler using a GCC 4.4 std library
- PR #1476[5323] - Adding `hpx::lcos::latch` and `hpx::lcos::local::latch`
- Issue #1475[5324] - Boost inspect violations
- PR #1473[5325] - Fixing action move tests
- Issue #1471[5326] - Sync primitives should not be movable
- PR #1470[5327] - Removing `hpx::util::polymorphic_factory`
- PR #1468[5328] - Fixed container creation
- Issue #1467[5329] - HPX application fail during finalization
- Issue #1466[5330] - HPX doesn't pick up Torque's nodefile on SuperMIC
- Issue #1464[5331] - HPX option for pre and post bootstrap performance counters
- PR #1463[5332] - Replacing `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1462[5333] - Consolidated task_region with N4411
- PR #1461[5334] - Consolidate inconsistent CMake option names
- Issue #1460[5335] - Which malloc is actually used? or at least which one is HPX built with
- Issue #1459[5336] - Make cmake configure step fail explicitly if compiler version is not supported
- Issue #1458[5337] - Update `parallel::task_region` with N4411

5315 https://github.com/STEllAR-GROUP/hpx/pull/1484
5316 https://github.com/STEllAR-GROUP/hpx/pull/1483
5317 https://github.com/STEllAR-GROUP/hpx/pull/1482
5318 https://github.com/STEllAR-GROUP/hpx/issues/1481
5319 https://github.com/STEllAR-GROUP/hpx/issues/1480
5320 https://github.com/STEllAR-GROUP/hpx/issues/1478
5321 https://github.com/STEllAR-GROUP/hpx/pull/1479
5322 https://github.com/STEllAR-GROUP/hpx/pull/1477
5323 https://github.com/STEllAR-GROUP/hpx/pull/1476
5324 https://github.com/STEllAR-GROUP/hpx/issues/1475
5325 https://github.com/STEllAR-GROUP/hpx/pull/1473
5326 https://github.com/STEllAR-GROUP/hpx/issues/1471
5327 https://github.com/STEllAR-GROUP/hpx/pull/1470
5328 https://github.com/STEllAR-GROUP/hpx/pull/1468
5329 https://github.com/STEllAR-GROUP/hpx/issues/1467
5330 https://github.com/STEllAR-GROUP/hpx/issues/1466
5331 https://github.com/STEllAR-GROUP/hpx/issues/1464
5332 https://github.com/STEllAR-GROUP/hpx/pull/1463
5333 https://github.com/STEllAR-GROUP/hpx/pull/1462
5334 https://github.com/STEllAR-GROUP/hpx/pull/1461
5335 https://github.com/STEllAR-GROUP/hpx/issues/1460
5336 https://github.com/STEllAR-GROUP/hpx/issues/1459
5337 https://github.com/STEllAR-GROUP/hpx/issues/1458

- PR #1456[5338] - Consolidating `new_<>()`
- Issue #1455[5339] - Replace `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1454[5340] - Removed harmful std::moves from return statements
- PR #1453[5341] - Use range-based for-loop instead of Boost.Foreach
- PR #1452[5342] - C++ feature tests
- PR #1451[5343] - When serializing, pass archive flags to traits::get_type_size
- Issue #1450[5344] - traits:get_type_size needs archive flags to enable zero_copy optimizations
- Issue #1449[5345] - "couldn't create performance counter" - AGAS
- Issue #1448[5346] - Replace distributing factories with `new_<T[]>(...)`
- PR #1447[5347] - Removing obsolete remote_object component
- PR #1446[5348] - Hpx serialization
- PR #1445[5349] - Replacing travis with circleci
- PR #1443[5350] - Always stripping HPX command line arguments before executing start function
- PR #1442[5351] - Adding –hpx:bind=none to disable thread affinities
- Issue #1439[5352] - Libraries get linked in multiple times, RPATH is not properly set
- PR #1438[5353] - Removed superfluous typedefs
- Issue #1437[5354] - `hpx::init()` should strip HPX-related flags from argv
- Issue #1436[5355] - Add strong scaling option to htts
- PR #1435[5356] - Adding `async_cb`, `async_continue_cb`, and `async_colocated_cb`
- PR #1434[5357] - Added missing install rule, removed some dead CMake code
- PR #1433[5358] - Add GitExternal and SubProject cmake scripts from eyescale/cmake repo
- Issue #1432[5359] - Add command line flag to disable thread pinning
- PR #1431[5360] - Fix #1423

[5338] https://github.com/STEllAR-GROUP/hpx/pull/1456
[5339] https://github.com/STEllAR-GROUP/hpx/issues/1455
[5340] https://github.com/STEllAR-GROUP/hpx/pull/1454
[5341] https://github.com/STEllAR-GROUP/hpx/pull/1453
[5342] https://github.com/STEllAR-GROUP/hpx/pull/1452
[5343] https://github.com/STEllAR-GROUP/hpx/pull/1451
[5344] https://github.com/STEllAR-GROUP/hpx/issues/1450
[5345] https://github.com/STEllAR-GROUP/hpx/issues/1449
[5346] https://github.com/STEllAR-GROUP/hpx/issues/1448
[5347] https://github.com/STEllAR-GROUP/hpx/pull/1447
[5348] https://github.com/STEllAR-GROUP/hpx/pull/1446
[5349] https://github.com/STEllAR-GROUP/hpx/pull/1445
[5350] https://github.com/STEllAR-GROUP/hpx/pull/1443
[5351] https://github.com/STEllAR-GROUP/hpx/pull/1442
[5352] https://github.com/STEllAR-GROUP/hpx/issues/1439
[5353] https://github.com/STEllAR-GROUP/hpx/pull/1438
[5354] https://github.com/STEllAR-GROUP/hpx/issues/1437
[5355] https://github.com/STEllAR-GROUP/hpx/issues/1436
[5356] https://github.com/STEllAR-GROUP/hpx/pull/1435
[5357] https://github.com/STEllAR-GROUP/hpx/pull/1434
[5358] https://github.com/STEllAR-GROUP/hpx/pull/1433
[5359] https://github.com/STEllAR-GROUP/hpx/issues/1432
[5360] https://github.com/STEllAR-GROUP/hpx/pull/1431

- Issue #1430[5361] - Inconsistent CMake option names
- Issue #1429[5362] - Configure setting `HPX_HAVE_PARCELPORT_MPI` is ignored
- PR #1428[5363] - Fixes #1419 (closed)
- PR #1427[5364] - Adding stencil_iterator and transform_iterator
- PR #1426[5365] - Fixes #1419
- PR #1425[5366] - During serialization memory allocation should honour allocator chunk size
- Issue #1424[5367] - chunk allocation during serialization does not use memory pool/allocator chunk size
- Issue #1423[5368] - Remove `HPX_STD_UNIQUE_PTR`
- Issue #1422[5369] - hpx:threads=all allocates too many os threads
- PR #1420[5370] - added .travis.yml
- Issue #1419[5371] - Unify enums: `hpx::runtime::state` and `hpx::state`
- PR #1416[5372] - Adding travis builder
- Issue #1414[5373] - Correct directory for dispatch_gcc46.hpp iteration
- Issue #1410[5374] - Set operation algorithms
- Issue #1389[5375] - Parallel algorithms relying on scan partitioner break for small number of elements
- Issue #1325[5376] - Exceptions thrown during parcel handling are not handled correctly
- Issue #1315[5377] - Errors while running performance tests
- Issue #1309[5378] - `hpx::vector` partitions are not easily extendable by applications
- PR #1300[5379] - Added serialization/de-serialization to examples.tuplespace
- Issue #1251[5380] - hpx::threads::get_thread_count doesn't consider pending threads
- Issue #1008[5381] - Decrease in application performance overtime; occasional spikes of major slowdown
- Issue #1001[5382] - Zero copy serialization raises assert
- Issue #721[5383] - Make HPX usable for Xeon Phi

5361 https://github.com/STEllAR-GROUP/hpx/issues/1430
5362 https://github.com/STEllAR-GROUP/hpx/issues/1429
5363 https://github.com/STEllAR-GROUP/hpx/pull/1428
5364 https://github.com/STEllAR-GROUP/hpx/pull/1427
5365 https://github.com/STEllAR-GROUP/hpx/pull/1426
5366 https://github.com/STEllAR-GROUP/hpx/pull/1425
5367 https://github.com/STEllAR-GROUP/hpx/issues/1424
5368 https://github.com/STEllAR-GROUP/hpx/issues/1423
5369 https://github.com/STEllAR-GROUP/hpx/issues/1422
5370 https://github.com/STEllAR-GROUP/hpx/pull/1420
5371 https://github.com/STEllAR-GROUP/hpx/issues/1419
5372 https://github.com/STEllAR-GROUP/hpx/pull/1416
5373 https://github.com/STEllAR-GROUP/hpx/issues/1414
5374 https://github.com/STEllAR-GROUP/hpx/issues/1410
5375 https://github.com/STEllAR-GROUP/hpx/issues/1389
5376 https://github.com/STEllAR-GROUP/hpx/issues/1325
5377 https://github.com/STEllAR-GROUP/hpx/issues/1315
5378 https://github.com/STEllAR-GROUP/hpx/issues/1309
5379 https://github.com/STEllAR-GROUP/hpx/pull/1300
5380 https://github.com/STEllAR-GROUP/hpx/issues/1251
5381 https://github.com/STEllAR-GROUP/hpx/issues/1008
5382 https://github.com/STEllAR-GROUP/hpx/issues/1001
5383 https://github.com/STEllAR-GROUP/hpx/issues/721

- Issue #524[5384] - Extend scheduler to support threads which can't be stolen

HPX V0.9.10 (Mar 24, 2015)

General changes

This is the 12th official release of *HPX*. It coincides with the 7th anniversary of the first commit to our source code repository. Since then, we have seen over 12300 commits amounting to more than 220000 lines of C++ code.

The major focus of this release was to improve the reliability of large scale runs. We believe to have achieved this goal as we now can reliably run *HPX* applications on up to ~24k cores. We have also shown that HPX can be used with success for symmetric runs (applications using both, host cores and Intel Xeon/Phi coprocessors). This is a huge step forward in terms of the usability of *HPX*. The main focus of this work involved isolating the causes of the segmentation faults at start up and shut down. Many of these issues were discovered to be the result of the suspension of threads which hold locks.

A very important improvement introduced with this release is the refactoring of the code representing our parcel-port implementation. Parcel- ports can now be implemented by 3rd parties as independent plugins which are dynamically loaded at runtime (static linking of parcel-ports is also supported). This refactoring also includes a massive improvement of the performance of our existing parcel-ports. We were able to significantly reduce the networking latencies and to improve the available networking bandwidth. Please note that in this release we disabled the ibverbs and ipc parcel ports as those have not been ported to the new plugin system yet (see Issue #839[5385]).

Another corner stone of this release is our work towards a complete implementation of __cpp11_n4104__ (Working Draft, Technical Specification for C++ Extensions for Parallelism). This document defines a set of parallel algorithms to be added to the C++ standard library. We now have implemented about 75% of all specified parallel algorithms (see [link hpx.manual.parallel.parallel_algorithms Parallel Algorithms] for more details). We also implemented some extensions to __cpp11_n4104__ allowing to invoke all of the algorithms asynchronously.

This release adds a first implementation of `hpx::vector` which is a distributed data structure closely aligned to the functionality of `std::vector`. The difference is that `hpx::vector` stores the data in partitions where the partitions can be distributed over different localities. We started to work on allowing to use the parallel algorithms with `hpx::vector`. At this point we have implemented only a few of the parallel algorithms to support distributed data structures (like `hpx::vector`) for testing purposes (see Issue #1338[5386] for a documentation of our progress).

Breaking changes

With this release we put a lot of effort into changing the code base to be more compatible to C++11. These changes have caused the following issues for backward compatibility:

- Move to Variadics- All of the API now uses variadic templates. However, this change required to modify the argument sequence for some of the exiting API functions (`hpx::async_continue`, `hpx::apply_continue`, `hpx::when_each`, `hpx::wait_each`, synchronous invocation of actions).
- Changes to Macros- We also removed the macros `HPX_STD_FUNCTION` and `HPX_STD_TUPLE`. This shouldn't affect any user code as we replaced `HPX_STD_FUNCTION` with `hpx::util::function_nonser` which was the default expansion used for this macro. All *HPX* API functions which expect a `hpx::util::function_nonser` (or a `hpx::util::unique_function_nonser`) can now be transparently called with a compatible `std::function` instead. Similarly, `HPX_STD_TUPLE` was replaced by its default expansion as well: `hpx::util::tuple`.

5384 https://github.com/STEllAR-GROUP/hpx/issues/524
5385 https://github.com/STEllAR-GROUP/hpx/issues/839
5386 https://github.com/STEllAR-GROUP/hpx/issues/1338

* Changes to `hpx::unique_future`- `hpx::unique_future`, which was deprecated in the previous release for `hpx::future` is now completely removed from *HPX*. This completes the transition to a completely standards conforming implementation of `hpx::future`.
* Changes to Supported Compilers. Finally, in order to utilize more C++11 semantics, we have officially dropped support for GCC 4.4 and MSVC 2012. Please see our *Prerequisites* page for more details.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

* Issue #1402[5387] - Internal shared_future serialization copies
* Issue #1399[5388] - Build takes unusually long time…
* Issue #1398[5389] - Tests using the scan partitioner are broken on at least gcc 4.7 and intel compiler
* Issue #1397[5390] - Completely remove hpx::unique_future
* Issue #1396[5391] - Parallel scan algorithms with different initial values
* Issue #1395[5392] - Race Condition - 1d_stencil_8 - SuperMIC
* Issue #1394[5393] - "suspending thread while at least one lock is being held" - 1d_stencil_8 - SuperMIC
* Issue #1393[5394] - SEGFAULT in 1d_stencil_8 on SuperMIC
* Issue #1392[5395] - Fixing #1168
* Issue #1391[5396] - Parallel Algorithms for scan partitioner for small number of elements
* Issue #1387[5397] - Failure with more than 4 localities
* Issue #1386[5398] - Dispatching unhandled exceptions to outer user code
* Issue #1385[5399] - Adding Copy algorithms, fixing `parallel::copy_if`
* Issue #1384[5400] - Fixing 1325
* Issue #1383[5401] - Fixed #504: Refactor Dataflow LCO to work with futures, this removes the dataflow component as it is obsolete
* Issue #1382[5402] - `is_sorted`, `is_sorted_until` and `is_partitioned` algorithms
* Issue #1381[5403] - fix for CMake versions prior to 3.1
* Issue #1380[5404] - resolved warning in CMake 3.1 and newer

5387 https://github.com/STEllAR-GROUP/hpx/issues/1402
5388 https://github.com/STEllAR-GROUP/hpx/issues/1399
5389 https://github.com/STEllAR-GROUP/hpx/issues/1398
5390 https://github.com/STEllAR-GROUP/hpx/issues/1397
5391 https://github.com/STEllAR-GROUP/hpx/issues/1396
5392 https://github.com/STEllAR-GROUP/hpx/issues/1395
5393 https://github.com/STEllAR-GROUP/hpx/issues/1394
5394 https://github.com/STEllAR-GROUP/hpx/issues/1393
5395 https://github.com/STEllAR-GROUP/hpx/issues/1392
5396 https://github.com/STEllAR-GROUP/hpx/issues/1391
5397 https://github.com/STEllAR-GROUP/hpx/issues/1387
5398 https://github.com/STEllAR-GROUP/hpx/issues/1386
5399 https://github.com/STEllAR-GROUP/hpx/issues/1385
5400 https://github.com/STEllAR-GROUP/hpx/issues/1384
5401 https://github.com/STEllAR-GROUP/hpx/issues/1383
5402 https://github.com/STEllAR-GROUP/hpx/issues/1382
5403 https://github.com/STEllAR-GROUP/hpx/issues/1381
5404 https://github.com/STEllAR-GROUP/hpx/issues/1380

- Issue #1379[5405] - Compilation error with papi
- Issue #1378[5406] - Towards safer migration
- Issue #1377[5407] - HPXConfig.cmake should include `TCMALLOC_LIBRARY` and `TCMALLOC_INCLUDE_DIR`
- Issue #1376[5408] - Warning on uninitialized member
- Issue #1375[5409] - Fixing 1163
- Issue #1374[5410] - Fixing the MSVC 12 release builder
- Issue #1373[5411] - Modifying parallel search algorithm for zero length searches
- Issue #1372[5412] - Modifying parallel search algorithm for zero length searches
- Issue #1371[5413] - Avoid holding a lock during agas::incref while doing a credit split
- Issue #1370[5414] - `--hpx:bind` throws unexpected error
- Issue #1369[5415] - Getting rid of (void) in loops
- Issue #1368[5416] - Variadic templates support for tuple
- Issue #1367[5417] - One last batch of variadic templates support
- Issue #1366[5418] - Fixing symbolic namespace hang
- Issue #1365[5419] - More held locks
- Issue #1364[5420] - Add counters 1363
- Issue #1363[5421] - Add thread overhead counters
- Issue #1362[5422] - Std config removal
- Issue #1361[5423] - Parcelport plugins
- Issue #1360[5424] - Detuplify transfer_action
- Issue #1359[5425] - Removed obsolete checks
- Issue #1358[5426] - Fixing 1352
- Issue #1357[5427] - Variadic templates support for runtime_support and components

5405 https://github.com/STEllAR-GROUP/hpx/issues/1379
5406 https://github.com/STEllAR-GROUP/hpx/issues/1378
5407 https://github.com/STEllAR-GROUP/hpx/issues/1377
5408 https://github.com/STEllAR-GROUP/hpx/issues/1376
5409 https://github.com/STEllAR-GROUP/hpx/issues/1375
5410 https://github.com/STEllAR-GROUP/hpx/issues/1374
5411 https://github.com/STEllAR-GROUP/hpx/issues/1373
5412 https://github.com/STEllAR-GROUP/hpx/issues/1372
5413 https://github.com/STEllAR-GROUP/hpx/issues/1371
5414 https://github.com/STEllAR-GROUP/hpx/issues/1370
5415 https://github.com/STEllAR-GROUP/hpx/issues/1369
5416 https://github.com/STEllAR-GROUP/hpx/issues/1368
5417 https://github.com/STEllAR-GROUP/hpx/issues/1367
5418 https://github.com/STEllAR-GROUP/hpx/issues/1366
5419 https://github.com/STEllAR-GROUP/hpx/issues/1365
5420 https://github.com/STEllAR-GROUP/hpx/issues/1364
5421 https://github.com/STEllAR-GROUP/hpx/issues/1363
5422 https://github.com/STEllAR-GROUP/hpx/issues/1362
5423 https://github.com/STEllAR-GROUP/hpx/issues/1361
5424 https://github.com/STEllAR-GROUP/hpx/issues/1360
5425 https://github.com/STEllAR-GROUP/hpx/issues/1359
5426 https://github.com/STEllAR-GROUP/hpx/issues/1358
5427 https://github.com/STEllAR-GROUP/hpx/issues/1357

- Issue #1356[5428] - fixed coordinate test for intel13
- Issue #1355[5429] - fixed coordinate.hpp
- Issue #1354[5430] - Lexicographical Compare completed
- Issue #1353[5431] - HPX should set `Boost_ADDITIONAL_VERSIONS` flags
- Issue #1352[5432] - Error: Cannot find action '' in type registry: HPX(bad_action_code)
- Issue #1351[5433] - Variadic templates support for appliers
- Issue #1350[5434] - Actions simplification
- Issue #1349[5435] - Variadic when and wait functions
- Issue #1348[5436] - Added hpx_init header to test files
- Issue #1347[5437] - Another batch of variadic templates support
- Issue #1346[5438] - Segmented copy
- Issue #1345[5439] - Attempting to fix hangs during shutdown
- Issue #1344[5440] - Std config removal
- Issue #1343[5441] - Removing various distribution policies for hpx::vector
- Issue #1342[5442] - Inclusive scan
- Issue #1341[5443] - Exclusive scan
- Issue #1340[5444] - Adding `parallel::count` for distributed data structures, adding tests
- Issue #1339[5445] - Update argument order for transform_reduce
- Issue #1337[5446] - Fix dataflow to handle properly ranges of futures
- Issue #1336[5447] - dataflow needs to hold onto futures passed to it
- Issue #1335[5448] - Fails to compile with msvc14
- Issue #1334[5449] - Examples build problem
- Issue #1333[5450] - Distributed transform reduce

5428 https://github.com/STEllAR-GROUP/hpx/issues/1356
5429 https://github.com/STEllAR-GROUP/hpx/issues/1355
5430 https://github.com/STEllAR-GROUP/hpx/issues/1354
5431 https://github.com/STEllAR-GROUP/hpx/issues/1353
5432 https://github.com/STEllAR-GROUP/hpx/issues/1352
5433 https://github.com/STEllAR-GROUP/hpx/issues/1351
5434 https://github.com/STEllAR-GROUP/hpx/issues/1350
5435 https://github.com/STEllAR-GROUP/hpx/issues/1349
5436 https://github.com/STEllAR-GROUP/hpx/issues/1348
5437 https://github.com/STEllAR-GROUP/hpx/issues/1347
5438 https://github.com/STEllAR-GROUP/hpx/issues/1346
5439 https://github.com/STEllAR-GROUP/hpx/issues/1345
5440 https://github.com/STEllAR-GROUP/hpx/issues/1344
5441 https://github.com/STEllAR-GROUP/hpx/issues/1343
5442 https://github.com/STEllAR-GROUP/hpx/issues/1342
5443 https://github.com/STEllAR-GROUP/hpx/issues/1341
5444 https://github.com/STEllAR-GROUP/hpx/issues/1340
5445 https://github.com/STEllAR-GROUP/hpx/issues/1339
5446 https://github.com/STEllAR-GROUP/hpx/issues/1337
5447 https://github.com/STEllAR-GROUP/hpx/issues/1336
5448 https://github.com/STEllAR-GROUP/hpx/issues/1335
5449 https://github.com/STEllAR-GROUP/hpx/issues/1334
5450 https://github.com/STEllAR-GROUP/hpx/issues/1333

- Issue #1332[5451] - Variadic templates support for actions
- Issue #1331[5452] - Some ambiguous calls of map::erase have been prevented by adding additional check in locality constructor.
- Issue #1330[5453] - Defining Plain Actions does not work as described in the documentation
- Issue #1329[5454] - Distributed vector cleanup
- Issue #1328[5455] - Sync docs and comments with code in hello_world example
- Issue #1327[5456] - Typos in docs
- Issue #1326[5457] - Documentation and code diverged in Fibonacci tutorial
- Issue #1325[5458] - Exceptions thrown during parcel handling are not handled correctly
- Issue #1324[5459] - fixed bandwidth calculation
- Issue #1323[5460] - mmap() failed to allocate thread stack due to insufficient resources
- Issue #1322[5461] - HPX fails to build aa182cf
- Issue #1321[5462] - Limiting size of outgoing messages while coalescing parcels
- Issue #1320[5463] - passing a future with launch::deferred in remote function call causes hang
- Issue #1319[5464] - An exception when tries to specify number high priority threads with abp-priority
- Issue #1318[5465] - Unable to run program with abp-priority and numa-sensitivity enabled
- Issue #1317[5466] - N4071 Search/Search_n finished, minor changes
- Issue #1316[5467] - Add config option to make -Ihpx.run_hpx_main!=1 the default
- Issue #1314[5468] - Variadic support for async and apply
- Issue #1313[5469] - Adjust when_any/some to the latest proposed interfaces
- Issue #1312[5470] - Fixing #857: hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #1311[5471] - Distributed get'er/set'er_values for distributed vector
- Issue #1310[5472] - Crashing in hpx::parcelset::policies::mpi::connection_handler::handle_messages() on SuperMIC

5451 https://github.com/STEllAR-GROUP/hpx/issues/1332
5452 https://github.com/STEllAR-GROUP/hpx/issues/1331
5453 https://github.com/STEllAR-GROUP/hpx/issues/1330
5454 https://github.com/STEllAR-GROUP/hpx/issues/1329
5455 https://github.com/STEllAR-GROUP/hpx/issues/1328
5456 https://github.com/STEllAR-GROUP/hpx/issues/1327
5457 https://github.com/STEllAR-GROUP/hpx/issues/1326
5458 https://github.com/STEllAR-GROUP/hpx/issues/1325
5459 https://github.com/STEllAR-GROUP/hpx/issues/1324
5460 https://github.com/STEllAR-GROUP/hpx/issues/1323
5461 https://github.com/STEllAR-GROUP/hpx/issues/1322
5462 https://github.com/STEllAR-GROUP/hpx/issues/1321
5463 https://github.com/STEllAR-GROUP/hpx/issues/1320
5464 https://github.com/STEllAR-GROUP/hpx/issues/1319
5465 https://github.com/STEllAR-GROUP/hpx/issues/1318
5466 https://github.com/STEllAR-GROUP/hpx/issues/1317
5467 https://github.com/STEllAR-GROUP/hpx/issues/1316
5468 https://github.com/STEllAR-GROUP/hpx/issues/1314
5469 https://github.com/STEllAR-GROUP/hpx/issues/1313
5470 https://github.com/STEllAR-GROUP/hpx/issues/1312
5471 https://github.com/STEllAR-GROUP/hpx/issues/1311
5472 https://github.com/STEllAR-GROUP/hpx/issues/1310

- Issue #1308[5473] - Unable to execute an application with –hpx:threads
- Issue #1307[5474] - merge_graph linking issue
- Issue #1306[5475] - First batch of variadic templates support
- Issue #1305[5476] - Create a compiler wrapper
- Issue #1304[5477] - Provide a compiler wrapper for hpx
- Issue #1303[5478] - Drop support for GCC44
- Issue #1302[5479] - Fixing #1297
- Issue #1301[5480] - Compilation error when tried to use boost range iterators with wait_all
- Issue #1298[5481] - Distributed vector
- Issue #1297[5482] - Unable to invoke component actions recursively
- Issue #1294[5483] - HDF5 build error
- Issue #1275[5484] - The parcelport implementation is non-optimal
- Issue #1267[5485] - Added classes and unit tests for local_file, orangefs_file and pxfs_file
- Issue #1264[5486] - Error "assertion '!m_fun' failed" randomly occurs when using TCP
- Issue #1254[5487] - thread binding seems to not work properly
- Issue #1220[5488] - parallel::copy_if is broken
- Issue #1217[5489] - Find a better way of fixing the issue patched by #1216
- Issue #1168[5490] - Starting HPX on Cray machines using aprun isn't working correctly
- Issue #1085[5491] - Replace startup and shutdown barriers with broadcasts
- Issue #981[5492] - With SLURM, –hpx:threads=8 should not be necessary
- Issue #857[5493] - hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #850[5494] - "flush" not documented
- Issue #763[5495] - Create buildbot instance that uses std::bind as HPX_STD_BIND

5473 https://github.com/STEllAR-GROUP/hpx/issues/1308
5474 https://github.com/STEllAR-GROUP/hpx/issues/1307
5475 https://github.com/STEllAR-GROUP/hpx/issues/1306
5476 https://github.com/STEllAR-GROUP/hpx/issues/1305
5477 https://github.com/STEllAR-GROUP/hpx/issues/1304
5478 https://github.com/STEllAR-GROUP/hpx/issues/1303
5479 https://github.com/STEllAR-GROUP/hpx/issues/1302
5480 https://github.com/STEllAR-GROUP/hpx/issues/1301
5481 https://github.com/STEllAR-GROUP/hpx/issues/1298
5482 https://github.com/STEllAR-GROUP/hpx/issues/1297
5483 https://github.com/STEllAR-GROUP/hpx/issues/1294
5484 https://github.com/STEllAR-GROUP/hpx/issues/1275
5485 https://github.com/STEllAR-GROUP/hpx/issues/1267
5486 https://github.com/STEllAR-GROUP/hpx/issues/1264
5487 https://github.com/STEllAR-GROUP/hpx/issues/1254
5488 https://github.com/STEllAR-GROUP/hpx/issues/1220
5489 https://github.com/STEllAR-GROUP/hpx/issues/1217
5490 https://github.com/STEllAR-GROUP/hpx/issues/1168
5491 https://github.com/STEllAR-GROUP/hpx/issues/1085
5492 https://github.com/STEllAR-GROUP/hpx/issues/981
5493 https://github.com/STEllAR-GROUP/hpx/issues/857
5494 https://github.com/STEllAR-GROUP/hpx/issues/850
5495 https://github.com/STEllAR-GROUP/hpx/issues/763

- Issue #680[5496] - Convert parcel ports into a plugin system
- Issue #582[5497] - Make exception thrown from HPX threads available from `hpx::init`
- Issue #504[5498] - Refactor Dataflow LCO to work with futures
- Issue #196[5499] - Don't store copies of the locality network metadata in the gva table

HPX V0.9.9 (Oct 31, 2014, codename Spooky)

General changes

We have had over 1500 commits since the last release and we have closed over 200 tickets (bugs, feature requests, pull requests, etc.). These are by far the largest numbers of commits and resolved issues for any of the *HPX* releases so far. We are especially happy about the large number of people who contributed for the first time to *HPX*.

- We completed the transition from the older (non-conforming) implementation of `hpx::future` to the new and fully conforming version by removing the old code and by renaming the type `hpx::unique_future` to `hpx::future`. In order to maintain backwards compatibility with existing code which uses the type `hpx::unique_future` we support the configuration variable `HPX_UNIQUE_FUTURE_ALIAS`. If this variable is set to `ON` while running cmake it will additionally define a template alias for this type.
- We rewrote and significantly changed our build system. Please have a look at the new (now generated) documentation here: *Building HPX*. Please revisit your build scripts to adapt to the changes. The most notable changes are:
  - `HPX_NO_INSTALL` is no longer necessary.
  - For external builds, you need to set `HPX_DIR` instead of `HPX_ROOT` as described here: *Using HPX with CMake-based projects*.
  - IDEs that support multiple configurations (Visual Studio and XCode) can now be used as intended. that means no build dir.
  - Building HPX statically (without dynamic libraries) is now supported (`-DHPX_STATIC_LINKING=On`).
  - Please note that many variables used to configure the build process have been renamed to unify the naming conventions (see the section *CMake options* for more information).
  - This also fixes a long list of issues, for more information see Issue #1204[5500].
- We started to implement various proposals to the C++ Standardization committee related to parallelism and concurrency, most notably N4409[5501] (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411[5502] (Task Region Rev. 3), and N4313[5503] (Working Draft, Technical Specification for C++ Extensions for Concurrency).
- We completely remodeled our automatic build system to run builds and unit tests on various systems and compilers. This allows us to find most bugs right as they were introduced and helps to maintain a high level of quality and compatibility. The newest build logs can be found at *HPX* Buildbot Website[5504].

[5496] https://github.com/STEllAR-GROUP/hpx/issues/680
[5497] https://github.com/STEllAR-GROUP/hpx/issues/582
[5498] https://github.com/STEllAR-GROUP/hpx/issues/504
[5499] https://github.com/STEllAR-GROUP/hpx/issues/196
[5500] https://github.com/STEllAR-GROUP/hpx/issues/1204
[5501] http://wg21.link/n4409
[5502] http://wg21.link/n4411
[5503] http://wg21.link/n4313
[5504] http://rostam.cct.lsu.edu/

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1296[5505] - Rename make_error_future to make_exceptional_future, adjust to N4123
- Issue #1295[5506] - building issue
- Issue #1293[5507] - Transpose example
- Issue #1292[5508] - Wrong abs() function used in example
- Issue #1291[5509] - non-synchronized shift operators have been removed
- Issue #1290[5510] - RDTSCP is defined as true for Xeon Phi build
- Issue #1289[5511] - Fixing 1288
- Issue #1288[5512] - Add new performance counters
- Issue #1287[5513] - Hierarchy scheduler broken performance counters
- Issue #1286[5514] - Algorithm cleanup
- Issue #1285[5515] - Broken Links in Documentation
- Issue #1284[5516] - Uninitialized copy
- Issue #1283[5517] - missing boost::scoped_ptr includes
- Issue #1282[5518] - Update documentation of build options for schedulers
- Issue #1281[5519] - reset idle rate counter
- Issue #1280[5520] - Bug when executing on Intel MIC
- Issue #1279[5521] - Add improved when_all/wait_all
- Issue #1278[5522] - Implement improved when_all/wait_all
- Issue #1277[5523] - feature request: get access to argc argv and variables_map
- Issue #1276[5524] - Remove merging map
- Issue #1274[5525] - Weird (wrong) string code in papi.cpp

5505 https://github.com/STEllAR-GROUP/hpx/issues/1296
5506 https://github.com/STEllAR-GROUP/hpx/issues/1295
5507 https://github.com/STEllAR-GROUP/hpx/issues/1293
5508 https://github.com/STEllAR-GROUP/hpx/issues/1292
5509 https://github.com/STEllAR-GROUP/hpx/issues/1291
5510 https://github.com/STEllAR-GROUP/hpx/issues/1290
5511 https://github.com/STEllAR-GROUP/hpx/issues/1289
5512 https://github.com/STEllAR-GROUP/hpx/issues/1288
5513 https://github.com/STEllAR-GROUP/hpx/issues/1287
5514 https://github.com/STEllAR-GROUP/hpx/issues/1286
5515 https://github.com/STEllAR-GROUP/hpx/issues/1285
5516 https://github.com/STEllAR-GROUP/hpx/issues/1284
5517 https://github.com/STEllAR-GROUP/hpx/issues/1283
5518 https://github.com/STEllAR-GROUP/hpx/issues/1282
5519 https://github.com/STEllAR-GROUP/hpx/issues/1281
5520 https://github.com/STEllAR-GROUP/hpx/issues/1280
5521 https://github.com/STEllAR-GROUP/hpx/issues/1279
5522 https://github.com/STEllAR-GROUP/hpx/issues/1278
5523 https://github.com/STEllAR-GROUP/hpx/issues/1277
5524 https://github.com/STEllAR-GROUP/hpx/issues/1276
5525 https://github.com/STEllAR-GROUP/hpx/issues/1274

- Issue #1273[5526] - Sequential task execution policy
- Issue #1272[5527] - Avoid CMake name clash for Boost.Thread library
- Issue #1271[5528] - Updates on HPX Test Units
- Issue #1270[5529] - hpx/util/safe_lexical_cast.hpp is added
- Issue #1269[5530] - Added default value for "LIB" cmake variable
- Issue #1268[5531] - Memory Counters not working
- Issue #1266[5532] - FindHPX.cmake is not installed
- Issue #1263[5533] - apply_remote test takes too long
- Issue #1262[5534] - Chrono cleanup
- Issue #1261[5535] - Need make install for papi counters and this builds all the examples
- Issue #1260[5536] - Documentation of Stencil example claims
- Issue #1259[5537] - Avoid double-linking Boost on Windows
- Issue #1257[5538] - Adding additional parameter to create_thread
- Issue #1256[5539] - added buildbot changes to release notes
- Issue #1255[5540] - Cannot build MiniGhost
- Issue #1253[5541] - hpx::thread defects
- Issue #1252[5542] - HPX_PREFIX is too fragile
- Issue #1250[5543] - switch_to_fiber_emulation does not work properly
- Issue #1249[5544] - Documentation is generated under Release folder
- Issue #1248[5545] - Fix usage of hpx_generic_coroutine_context and get tests passing on powerpc
- Issue #1247[5546] - Dynamic linking error
- Issue #1246[5547] - Make cpuid.cpp C++11 compliant
- Issue #1245[5548] - HPX fails on startup (setting thread affinity mask)

[5526] https://github.com/STEllAR-GROUP/hpx/issues/1273
[5527] https://github.com/STEllAR-GROUP/hpx/issues/1272
[5528] https://github.com/STEllAR-GROUP/hpx/issues/1271
[5529] https://github.com/STEllAR-GROUP/hpx/issues/1270
[5530] https://github.com/STEllAR-GROUP/hpx/issues/1269
[5531] https://github.com/STEllAR-GROUP/hpx/issues/1268
[5532] https://github.com/STEllAR-GROUP/hpx/issues/1266
[5533] https://github.com/STEllAR-GROUP/hpx/issues/1263
[5534] https://github.com/STEllAR-GROUP/hpx/issues/1262
[5535] https://github.com/STEllAR-GROUP/hpx/issues/1261
[5536] https://github.com/STEllAR-GROUP/hpx/issues/1260
[5537] https://github.com/STEllAR-GROUP/hpx/issues/1259
[5538] https://github.com/STEllAR-GROUP/hpx/issues/1257
[5539] https://github.com/STEllAR-GROUP/hpx/issues/1256
[5540] https://github.com/STEllAR-GROUP/hpx/issues/1255
[5541] https://github.com/STEllAR-GROUP/hpx/issues/1253
[5542] https://github.com/STEllAR-GROUP/hpx/issues/1252
[5543] https://github.com/STEllAR-GROUP/hpx/issues/1250
[5544] https://github.com/STEllAR-GROUP/hpx/issues/1249
[5545] https://github.com/STEllAR-GROUP/hpx/issues/1248
[5546] https://github.com/STEllAR-GROUP/hpx/issues/1247
[5547] https://github.com/STEllAR-GROUP/hpx/issues/1246
[5548] https://github.com/STEllAR-GROUP/hpx/issues/1245

- Issue #1244[5549] - HPX_WITH_RDTSC configure test fails, but should succeed
- Issue #1243[5550] - CTest dashboard info for CSCS CDash drop location
- Issue #1242[5551] - Mac fixes
- Issue #1241[5552] - Failure in Distributed with Boost 1.56
- Issue #1240[5553] - fix a race condition in examples.diskperf
- Issue #1239[5554] - fix wait_each in examples.diskperf
- Issue #1238[5555] - Fixed #1237: hpx::util::portable_binary_iarchive failed
- Issue #1237[5556] - hpx::util::portable_binary_iarchive faileds
- Issue #1235[5557] - Fixing clang warnings and errors
- Issue #1234[5558] - TCP runs fail: Transport endpoint is not connected
- Issue #1233[5559] - Making sure the correct number of threads is registered with AGAS
- Issue #1232[5560] - Fixing race in wait_xxx
- Issue #1231[5561] - Parallel minmax
- Issue #1230[5562] - Distributed run of 1d_stencil_8 uses less threads than spec. & sometimes gives errors
- Issue #1229[5563] - Unstable number of threads
- Issue #1228[5564] - HPX link error (cmake / MPI)
- Issue #1226[5565] - Warning about struct/class thread_counters
- Issue #1225[5566] - Adding parallel::replace etc
- Issue #1224[5567] - Extending dataflow to pass through non-future arguments
- Issue #1223[5568] - Remaining find algorithms implemented, N4071
- Issue #1222[5569] - Merging all the changes
- Issue #1221[5570] - No error output when using mpirun with hpx
- Issue #1219[5571] - Adding new AGAS cache performance counters

[5549] https://github.com/STEllAR-GROUP/hpx/issues/1244
[5550] https://github.com/STEllAR-GROUP/hpx/issues/1243
[5551] https://github.com/STEllAR-GROUP/hpx/issues/1242
[5552] https://github.com/STEllAR-GROUP/hpx/issues/1241
[5553] https://github.com/STEllAR-GROUP/hpx/issues/1240
[5554] https://github.com/STEllAR-GROUP/hpx/issues/1239
[5555] https://github.com/STEllAR-GROUP/hpx/issues/1238
[5556] https://github.com/STEllAR-GROUP/hpx/issues/1237
[5557] https://github.com/STEllAR-GROUP/hpx/issues/1235
[5558] https://github.com/STEllAR-GROUP/hpx/issues/1234
[5559] https://github.com/STEllAR-GROUP/hpx/issues/1233
[5560] https://github.com/STEllAR-GROUP/hpx/issues/1232
[5561] https://github.com/STEllAR-GROUP/hpx/issues/1231
[5562] https://github.com/STEllAR-GROUP/hpx/issues/1230
[5563] https://github.com/STEllAR-GROUP/hpx/issues/1229
[5564] https://github.com/STEllAR-GROUP/hpx/issues/1228
[5565] https://github.com/STEllAR-GROUP/hpx/issues/1226
[5566] https://github.com/STEllAR-GROUP/hpx/issues/1225
[5567] https://github.com/STEllAR-GROUP/hpx/issues/1224
[5568] https://github.com/STEllAR-GROUP/hpx/issues/1223
[5569] https://github.com/STEllAR-GROUP/hpx/issues/1222
[5570] https://github.com/STEllAR-GROUP/hpx/issues/1221
[5571] https://github.com/STEllAR-GROUP/hpx/issues/1219

- Issue #1216[5572] - Fixing using futures (clients) as arguments to actions
- Issue #1215[5573] - Error compiling simple component
- Issue #1214[5574] - Stencil docs
- Issue #1213[5575] - Using more than a few dozen MPI processes on SuperMike results in a seg fault before getting to hpx_main
- Issue #1212[5576] - Parallel rotate
- Issue #1211[5577] - Direct actions cause the future's shared_state to be leaked
- Issue #1210[5578] - Refactored local::promise to be standard conformant
- Issue #1209[5579] - Improve command line handling
- Issue #1208[5580] - Adding parallel::reverse and parallel::reverse_copy
- Issue #1207[5581] - Add copy_backward and move_backward
- Issue #1206[5582] - N4071 additional algorithms implemented
- Issue #1204[5583] - Cmake simplification and various other minor changes
- Issue #1203[5584] - Implementing new launch policy for (local) async: `hpx::launch::fork`.
- Issue #1202[5585] - Failed assertion in connection_cache.hpp
- Issue #1201[5586] - pkg-config doesn't add mpi link directories
- Issue #1200[5587] - Error when querying time performance counters
- Issue #1199[5588] - library path is now configurable (again)
- Issue #1198[5589] - Error when querying performance counters
- Issue #1197[5590] - tests fail with intel compiler
- Issue #1196[5591] - Silence several warnings
- Issue #1195[5592] - Rephrase initializers to work with VC++ 2012
- Issue #1194[5593] - Simplify parallel algorithms
- Issue #1193[5594] - Adding `parallel::equal`

5572 https://github.com/STEllAR-GROUP/hpx/issues/1216
5573 https://github.com/STEllAR-GROUP/hpx/issues/1215
5574 https://github.com/STEllAR-GROUP/hpx/issues/1214
5575 https://github.com/STEllAR-GROUP/hpx/issues/1213
5576 https://github.com/STEllAR-GROUP/hpx/issues/1212
5577 https://github.com/STEllAR-GROUP/hpx/issues/1211
5578 https://github.com/STEllAR-GROUP/hpx/issues/1210
5579 https://github.com/STEllAR-GROUP/hpx/issues/1209
5580 https://github.com/STEllAR-GROUP/hpx/issues/1208
5581 https://github.com/STEllAR-GROUP/hpx/issues/1207
5582 https://github.com/STEllAR-GROUP/hpx/issues/1206
5583 https://github.com/STEllAR-GROUP/hpx/issues/1204
5584 https://github.com/STEllAR-GROUP/hpx/issues/1203
5585 https://github.com/STEllAR-GROUP/hpx/issues/1202
5586 https://github.com/STEllAR-GROUP/hpx/issues/1201
5587 https://github.com/STEllAR-GROUP/hpx/issues/1200
5588 https://github.com/STEllAR-GROUP/hpx/issues/1199
5589 https://github.com/STEllAR-GROUP/hpx/issues/1198
5590 https://github.com/STEllAR-GROUP/hpx/issues/1197
5591 https://github.com/STEllAR-GROUP/hpx/issues/1196
5592 https://github.com/STEllAR-GROUP/hpx/issues/1195
5593 https://github.com/STEllAR-GROUP/hpx/issues/1194
5594 https://github.com/STEllAR-GROUP/hpx/issues/1193

- Issue #1192[5595] - HPX(out_of_memory) on including <hpx/hpx.hpp>
- Issue #1191[5596] - Fixing #1189
- Issue #1190[5597] - Chrono cleanup
- Issue #1189[5598] - Deadlock .. somewhere? (probably serialization)
- Issue #1188[5599] - Removed `future::get_status()`
- Issue #1186[5600] - Fixed FindOpenCL to find current AMD APP SDK
- Issue #1184[5601] - Tweaking future unwrapping
- Issue #1183[5602] - Extended `parallel::reduce`
- Issue #1182[5603] - `future::unwrap` hangs for `launch::deferred`
- Issue #1181[5604] - Adding `all_of`, `any_of`, and `none_of` and corresponding documentation
- Issue #1180[5605] - `hpx::cout` defect
- Issue #1179[5606] - `hpx::async` does not work for member function pointers when called on types with self-defined unary `operator*`
- Issue #1178[5607] - Implemented variadic `hpx::util::zip_iterator`
- Issue #1177[5608] - MPI parcelport defect
- Issue #1176[5609] - `HPX_DEFINE_COMPONENT_CONST_ACTION_TPL` does not have a 2-argument version
- Issue #1175[5610] - Create util::zip_iterator working with util::tuple<>
- Issue #1174[5611] - Error Building HPX on linux, root_certificate_authority.cpp
- Issue #1173[5612] - hpx::cout output lost
- Issue #1172[5613] - HPX build error with Clang 3.4.2
- Issue #1171[5614] - `CMAKE_INSTALL_PREFIX` ignored
- Issue #1170[5615] - Close hpx_benchmarks repository on Github
- Issue #1169[5616] - Buildbot emails have syntax error in url
- Issue #1167[5617] - Merge partial implementation of standards proposal N3960

5595 https://github.com/STEllAR-GROUP/hpx/issues/1192
5596 https://github.com/STEllAR-GROUP/hpx/issues/1191
5597 https://github.com/STEllAR-GROUP/hpx/issues/1190
5598 https://github.com/STEllAR-GROUP/hpx/issues/1189
5599 https://github.com/STEllAR-GROUP/hpx/issues/1188
5600 https://github.com/STEllAR-GROUP/hpx/issues/1186
5601 https://github.com/STEllAR-GROUP/hpx/issues/1184
5602 https://github.com/STEllAR-GROUP/hpx/issues/1183
5603 https://github.com/STEllAR-GROUP/hpx/issues/1182
5604 https://github.com/STEllAR-GROUP/hpx/issues/1181
5605 https://github.com/STEllAR-GROUP/hpx/issues/1180
5606 https://github.com/STEllAR-GROUP/hpx/issues/1179
5607 https://github.com/STEllAR-GROUP/hpx/issues/1178
5608 https://github.com/STEllAR-GROUP/hpx/issues/1177
5609 https://github.com/STEllAR-GROUP/hpx/issues/1176
5610 https://github.com/STEllAR-GROUP/hpx/issues/1175
5611 https://github.com/STEllAR-GROUP/hpx/issues/1174
5612 https://github.com/STEllAR-GROUP/hpx/issues/1173
5613 https://github.com/STEllAR-GROUP/hpx/issues/1172
5614 https://github.com/STEllAR-GROUP/hpx/issues/1171
5615 https://github.com/STEllAR-GROUP/hpx/issues/1170
5616 https://github.com/STEllAR-GROUP/hpx/issues/1169
5617 https://github.com/STEllAR-GROUP/hpx/issues/1167

- Issue #1166[5618] - Fixed several compiler warnings
- Issue #1165[5619] - cmake warns: "tests.regressions.actions" does not exist
- Issue #1164[5620] - Want my own serialization of hpx::future
- Issue #1162[5621] - Segfault in hello_world example
- Issue #1161[5622] - Use `HPX_ASSERT` to aid the compiler
- Issue #1160[5623] - Do not put -DNDEBUG into hpx_application.pc
- Issue #1159[5624] - Support Clang 3.4.2
- Issue #1158[5625] - Fixed #1157: Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1157[5626] - Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1156[5627] - Force inlining fails
- Issue #1155[5628] - changed header of printout to be compatible with python csv module
- Issue #1154[5629] - Fixing iostreams
- Issue #1153[5630] - Standard manipulators (like std::endl) do not work with hpx::ostream
- Issue #1152[5631] - Functions revamp
- Issue #1151[5632] - Suppressing cmake 3.0 policy warning for CMP0026
- Issue #1150[5633] - Client Serialization error
- Issue #1149[5634] - Segfault on Stampede
- Issue #1148[5635] - Refactoring mini-ghost
- Issue #1147[5636] - N3960 copy_if and copy_n implemented and tested
- Issue #1146[5637] - Stencil print
- Issue #1145[5638] - N3960 hpx::parallel::copy implemented and tested
- Issue #1144[5639] - OpenMP examples 1d_stencil do not build
- Issue #1143[5640] - 1d_stencil OpenMP examples do not build

[5618] https://github.com/STEllAR-GROUP/hpx/issues/1166
[5619] https://github.com/STEllAR-GROUP/hpx/issues/1165
[5620] https://github.com/STEllAR-GROUP/hpx/issues/1164
[5621] https://github.com/STEllAR-GROUP/hpx/issues/1162
[5622] https://github.com/STEllAR-GROUP/hpx/issues/1161
[5623] https://github.com/STEllAR-GROUP/hpx/issues/1160
[5624] https://github.com/STEllAR-GROUP/hpx/issues/1159
[5625] https://github.com/STEllAR-GROUP/hpx/issues/1158
[5626] https://github.com/STEllAR-GROUP/hpx/issues/1157
[5627] https://github.com/STEllAR-GROUP/hpx/issues/1156
[5628] https://github.com/STEllAR-GROUP/hpx/issues/1155
[5629] https://github.com/STEllAR-GROUP/hpx/issues/1154
[5630] https://github.com/STEllAR-GROUP/hpx/issues/1153
[5631] https://github.com/STEllAR-GROUP/hpx/issues/1152
[5632] https://github.com/STEllAR-GROUP/hpx/issues/1151
[5633] https://github.com/STEllAR-GROUP/hpx/issues/1150
[5634] https://github.com/STEllAR-GROUP/hpx/issues/1149
[5635] https://github.com/STEllAR-GROUP/hpx/issues/1148
[5636] https://github.com/STEllAR-GROUP/hpx/issues/1147
[5637] https://github.com/STEllAR-GROUP/hpx/issues/1146
[5638] https://github.com/STEllAR-GROUP/hpx/issues/1145
[5639] https://github.com/STEllAR-GROUP/hpx/issues/1144
[5640] https://github.com/STEllAR-GROUP/hpx/issues/1143

- Issue #1142[5641] - Cannot build HPX with gcc 4.6 on OS X
- Issue #1140[5642] - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
- Issue #1139[5643] - hpx/hpx/config/compiler_specific.hpp
- Issue #1138[5644] - clean up pkg-config files
- Issue #1137[5645] - Improvements to create binary packages
- Issue #1136[5646] - HPX_GCC_VERSION not defined on all compilers
- Issue #1135[5647] - Avoiding collision between winsock2.h and windows.h
- Issue #1134[5648] - Making sure, that hpx::finalize can be called from any locality
- Issue #1133[5649] - 1d stencil examples
- Issue #1131[5650] - Refactor unique_function implementation
- Issue #1130[5651] - Unique function
- Issue #1129[5652] - Some fixes to the Build system on OS X
- Issue #1128[5653] - Action future args
- Issue #1127[5654] - Executor causes segmentation fault
- Issue #1124[5655] - Adding new API functions: `register_id_with_basename`, `unregister_id_with_basename`, `find_ids_from_basename`; adding test
- Issue #1123[5656] - Reduce nesting of try-catch construct in `encode_parcels`?
- Issue #1122[5657] - Client base fixes
- Issue #1121[5658] - Update `hpxrun.py.in`
- Issue #1120[5659] - HTTS2 tests compile errors on v110 (VS2012)
- Issue #1119[5660] - Remove references to boost::atomic in accumulator example
- Issue #1118[5661] - Only build test thread_pool_executor_1114_test if `HPX_SCHEDULER` is set
- Issue #1117[5662] - local_queue_executor linker error on vc110
- Issue #1116[5663] - Disabled performance counter should give runtime errors, not invalid data

[5641] https://github.com/STEllAR-GROUP/hpx/issues/1142
[5642] https://github.com/STEllAR-GROUP/hpx/issues/1140
[5643] https://github.com/STEllAR-GROUP/hpx/issues/1139
[5644] https://github.com/STEllAR-GROUP/hpx/issues/1138
[5645] https://github.com/STEllAR-GROUP/hpx/issues/1137
[5646] https://github.com/STEllAR-GROUP/hpx/issues/1136
[5647] https://github.com/STEllAR-GROUP/hpx/issues/1135
[5648] https://github.com/STEllAR-GROUP/hpx/issues/1134
[5649] https://github.com/STEllAR-GROUP/hpx/issues/1133
[5650] https://github.com/STEllAR-GROUP/hpx/issues/1131
[5651] https://github.com/STEllAR-GROUP/hpx/issues/1130
[5652] https://github.com/STEllAR-GROUP/hpx/issues/1129
[5653] https://github.com/STEllAR-GROUP/hpx/issues/1128
[5654] https://github.com/STEllAR-GROUP/hpx/issues/1127
[5655] https://github.com/STEllAR-GROUP/hpx/issues/1124
[5656] https://github.com/STEllAR-GROUP/hpx/issues/1123
[5657] https://github.com/STEllAR-GROUP/hpx/issues/1122
[5658] https://github.com/STEllAR-GROUP/hpx/issues/1121
[5659] https://github.com/STEllAR-GROUP/hpx/issues/1120
[5660] https://github.com/STEllAR-GROUP/hpx/issues/1119
[5661] https://github.com/STEllAR-GROUP/hpx/issues/1118
[5662] https://github.com/STEllAR-GROUP/hpx/issues/1117
[5663] https://github.com/STEllAR-GROUP/hpx/issues/1116

- Issue #1115[5664] - Compile error with Intel C++ 13.1
- Issue #1114[5665] - Default constructed executor is not usable
- Issue #1113[5666] - Fast compilation of logging causes ABI incompatibilities between different `NDEBUG` values
- Issue #1112[5667] - Using thread_pool_executors causes segfault
- Issue #1111[5668] - `hpx::threads::get_thread_data` always returns zero
- Issue #1110[5669] - Remove unnecessary null pointer checks
- Issue #1109[5670] - More tests adjustments
- Issue #1108[5671] - Clarify build rules for "libboost_atomic-mt.so"?
- Issue #1107[5672] - Remove unnecessary null pointer checks
- Issue #1106[5673] - network_storage benchmark improvements, adding legends to plots and tidying layout
- Issue #1105[5674] - Add more plot outputs and improve instructions doc
- Issue #1104[5675] - Complete quoting for parameters of some CMake commands
- Issue #1103[5676] - Work on test/scripts
- Issue #1102[5677] - Changed minimum requirement of window install to 2012
- Issue #1101[5678] - Changed minimum requirement of window install to 2012
- Issue #1100[5679] - Changed readme to no longer specify using MSVC 2010 compiler
- Issue #1099[5680] - Error returning futures from component actions
- Issue #1098[5681] - Improve storage test
- Issue #1097[5682] - data_actions quickstart example calls missing function decorate_action of data_get_action
- Issue #1096[5683] - MPI parcelport broken with new zero copy optimization
- Issue #1095[5684] - Warning C4005: _WIN32_WINNT: Macro redefinition
- Issue #1094[5685] - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
- Issue #1093[5686] - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS

5664 https://github.com/STEllAR-GROUP/hpx/issues/1115
5665 https://github.com/STEllAR-GROUP/hpx/issues/1114
5666 https://github.com/STEllAR-GROUP/hpx/issues/1113
5667 https://github.com/STEllAR-GROUP/hpx/issues/1112
5668 https://github.com/STEllAR-GROUP/hpx/issues/1111
5669 https://github.com/STEllAR-GROUP/hpx/issues/1110
5670 https://github.com/STEllAR-GROUP/hpx/issues/1109
5671 https://github.com/STEllAR-GROUP/hpx/issues/1108
5672 https://github.com/STEllAR-GROUP/hpx/issues/1107
5673 https://github.com/STEllAR-GROUP/hpx/issues/1106
5674 https://github.com/STEllAR-GROUP/hpx/issues/1105
5675 https://github.com/STEllAR-GROUP/hpx/issues/1104
5676 https://github.com/STEllAR-GROUP/hpx/issues/1103
5677 https://github.com/STEllAR-GROUP/hpx/issues/1102
5678 https://github.com/STEllAR-GROUP/hpx/issues/1101
5679 https://github.com/STEllAR-GROUP/hpx/issues/1100
5680 https://github.com/STEllAR-GROUP/hpx/issues/1099
5681 https://github.com/STEllAR-GROUP/hpx/issues/1098
5682 https://github.com/STEllAR-GROUP/hpx/issues/1097
5683 https://github.com/STEllAR-GROUP/hpx/issues/1096
5684 https://github.com/STEllAR-GROUP/hpx/issues/1095
5685 https://github.com/STEllAR-GROUP/hpx/issues/1094
5686 https://github.com/STEllAR-GROUP/hpx/issues/1093

- Issue #1092[5687] - Rename unique_future<> back to future<>
- Issue #1091[5688] - Inconsistent error message
- Issue #1090[5689] - On windows 8.1 the examples crashed if using more than one os thread
- Issue #1089[5690] - Components should be allowed to have their own executor
- Issue #1088[5691] - Add possibility to select a network interface for the ibverbs parcelport
- Issue #1087[5692] - ibverbs and ipc parcelport uses zero copy optimization
- Issue #1083[5693] - Make shell examples copyable in docs
- Issue #1082[5694] - Implement proper termination detection during shutdown
- Issue #1081[5695] - Implement thread_specific_ptr for hpx::threads
- Issue #1072[5696] - make install not working properly
- Issue #1070[5697] - Complete quoting for parameters of some CMake commands
- Issue #1059[5698] - Fix more unused variable warnings
- Issue #1051[5699] - Implement when_each
- Issue #973[5700] - Would like option to report hwloc bindings
- Issue #970[5701] - Bad flags for Fortran compiler
- Issue #941[5702] - Create a proper user level context switching class for BG/Q
- Issue #935[5703] - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #934[5704] - Want to build HPX without dynamic libraries
- Issue #927[5705] - Make hpx/lcos/reduce.hpp accept futures of id_type
- Issue #926[5706] - All unit tests that are run with more than one thread with CTest/hpx_run_test should configure hpx.os_threads
- Issue #925[5707] - regression_dataflow_791 needs to be brought in line with HPX standards
- Issue #899[5708] - Fix race conditions in regression tests
- Issue #879[5709] - Hung test leads to cascading test failure; make tests should support the MPI parcelport

[5687] https://github.com/STEllAR-GROUP/hpx/issues/1092
[5688] https://github.com/STEllAR-GROUP/hpx/issues/1091
[5689] https://github.com/STEllAR-GROUP/hpx/issues/1090
[5690] https://github.com/STEllAR-GROUP/hpx/issues/1089
[5691] https://github.com/STEllAR-GROUP/hpx/issues/1088
[5692] https://github.com/STEllAR-GROUP/hpx/issues/1087
[5693] https://github.com/STEllAR-GROUP/hpx/issues/1083
[5694] https://github.com/STEllAR-GROUP/hpx/issues/1082
[5695] https://github.com/STEllAR-GROUP/hpx/issues/1081
[5696] https://github.com/STEllAR-GROUP/hpx/issues/1072
[5697] https://github.com/STEllAR-GROUP/hpx/issues/1070
[5698] https://github.com/STEllAR-GROUP/hpx/issues/1059
[5699] https://github.com/STEllAR-GROUP/hpx/issues/1051
[5700] https://github.com/STEllAR-GROUP/hpx/issues/973
[5701] https://github.com/STEllAR-GROUP/hpx/issues/970
[5702] https://github.com/STEllAR-GROUP/hpx/issues/941
[5703] https://github.com/STEllAR-GROUP/hpx/issues/935
[5704] https://github.com/STEllAR-GROUP/hpx/issues/934
[5705] https://github.com/STEllAR-GROUP/hpx/issues/927
[5706] https://github.com/STEllAR-GROUP/hpx/issues/926
[5707] https://github.com/STEllAR-GROUP/hpx/issues/925
[5708] https://github.com/STEllAR-GROUP/hpx/issues/899
[5709] https://github.com/STEllAR-GROUP/hpx/issues/879

- Issue #865[5710] - future<T> and friends shall work for movable only Ts
- Issue #847[5711] - Dynamic libraries are not installed on OS X
- Issue #816[5712] - First Program tutorial pull request
- Issue #799[5713] - Wrap lexical_cast to avoid exceptions
- Issue #720[5714] - broken configuration when using ccmake on Ubuntu
- Issue #622[5715] - `--hpx:hpx` and `--hpx:debug-hpx-log` is nonsensical
- Issue #525[5716] - Extend barrier LCO test to run in distributed
- Issue #515[5717] - Multi-destination version of hpx::apply is broken
- Issue #509[5718] - Push Boost.Atomic changes upstream
- Issue #503[5719] - Running HPX applications on Windows should not require setting %PATH%
- Issue #461[5720] - Add a compilation sanity test
- Issue #456[5721] - hpx_run_tests.py should log output from tests that timeout
- Issue #454[5722] - Investigate threadmanager performance
- Issue #345[5723] - Add more versatile environmental/cmake variable support to hpx_find_* CMake macros
- Issue #209[5724] - Support multiple configurations in generated build files
- Issue #190[5725] - hpx::cout should be a std::ostream
- Issue #189[5726] - iostreams component should use startup/shutdown functions
- Issue #183[5727] - Use Boost.ICL for correctness in AGAS
- Issue #44[5728] - Implement real futures

[5710] https://github.com/STEllAR-GROUP/hpx/issues/865
[5711] https://github.com/STEllAR-GROUP/hpx/issues/847
[5712] https://github.com/STEllAR-GROUP/hpx/issues/816
[5713] https://github.com/STEllAR-GROUP/hpx/issues/799
[5714] https://github.com/STEllAR-GROUP/hpx/issues/720
[5715] https://github.com/STEllAR-GROUP/hpx/issues/622
[5716] https://github.com/STEllAR-GROUP/hpx/issues/525
[5717] https://github.com/STEllAR-GROUP/hpx/issues/515
[5718] https://github.com/STEllAR-GROUP/hpx/issues/509
[5719] https://github.com/STEllAR-GROUP/hpx/issues/503
[5720] https://github.com/STEllAR-GROUP/hpx/issues/461
[5721] https://github.com/STEllAR-GROUP/hpx/issues/456
[5722] https://github.com/STEllAR-GROUP/hpx/issues/454
[5723] https://github.com/STEllAR-GROUP/hpx/issues/345
[5724] https://github.com/STEllAR-GROUP/hpx/issues/209
[5725] https://github.com/STEllAR-GROUP/hpx/issues/190
[5726] https://github.com/STEllAR-GROUP/hpx/issues/189
[5727] https://github.com/STEllAR-GROUP/hpx/issues/183
[5728] https://github.com/STEllAR-GROUP/hpx/issues/44

HPX V0.9.8 (Mar 24, 2014)

We have had over 800 commits since the last release and we have closed over 65 tickets (bugs, feature requests, etc.).

With the changes below, *HPX* is once again leading the charge of a whole new era of computation. By intrinsically breaking down and synchronizing the work to be done, *HPX* insures that application developers will no longer have to fret about where a segment of code executes. That allows coders to focus their time and energy to understanding the data dependencies of their algorithms and thereby the core obstacles to an efficient code. Here are some of the advantages of using *HPX*:

- *HPX* is solidly rooted in a sophisticated theoretical execution model – ParalleX
- *HPX* exposes an API fully conforming to the C++11 and the draft C++14 standards, extended and applied to distributed computing. Everything programmers know about the concurrency primitives of the standard C++ library is still valid in the context of *HPX*.
- It provides a competitive, high performance implementation of modern, future-proof ideas which gives an smooth migration path from today's mainstream techniques
- There is no need for the programmer to worry about lower level parallelization paradigms like threads or message passing; no need to understand pthreads, MPI, OpenMP, or Windows threads, etc.
- There is no need to think about different types of parallelism such as tasks, pipelines, or fork-join, task or data parallelism.
- The same source of your program compiles and runs on Linux, BlueGene/Q, Mac OS X, Windows, and Android.
- The same code runs on shared memory multi-core systems and supercomputers, on handheld devices and Intel® Xeon Phi™ accelerators, or a heterogeneous mix of those.

General changes

- A major API breaking change for this release was introduced by implementing `hpx::future` and `hpx::shared_future` fully in conformance with the C++11 Standard[5729]. While `hpx::shared_future` is new and will not create any compatibility problems, we revised the interface and implementation of the existing `hpx::future`. For more details please see the mailing list archive[5730]. To avoid any incompatibilities for existing code we named the type which implements the `std::future` interface as `hpx::unique_future`. For the next release this will be renamed to `hpx::future`, making it full conforming to C++11 Standard[5731].
- A large part of the code base of *HPX* has been refactored and partially re-implemented. The main changes were related to
  - The threading subsystem: these changes significantly reduce the amount of overheads caused by the schedulers, improve the modularity of the code base, and extend the variety of available scheduling algorithms.
  - The parcel subsystem: these changes improve the performance of the *HPX* networking layer, modularize the structure of the parcelports, and simplify the creation of new parcelports for other underlying networking libraries.
  - The API subsystem: these changes improved the conformance of the API to C++11 Standard, extend and unify the available API functionality, and decrease the overheads created by various elements of the API.
  - The robustness of the component loading subsystem has been improved significantly, allowing to more portably and more reliably register the components needed by an application as startup. This additionally speeds up general application initialization.

[5729] http://www.open-std.org/jtc1/sc22/wg21
[5730] http://mail.cct.lsu.edu/pipermail/hpx-users/2014-January/000141.html
[5731] http://www.open-std.org/jtc1/sc22/wg21

- We added new API functionality like `hpx::migrate` and `hpx::copy_component` which are the basic building blocks necessary for implementing higher level abstractions for system-wide load balancing, runtime-adaptive resource management, and object-oriented checkpointing and state-management.
- We removed the use of C++11 move emulation (using Boost.Move), replacing it with C++11 rvalue references. This is the first step towards using more and more native C++11 facilities which we plan to introduce in the future.
- We improved the reference counting scheme used by *HPX* which helps managing distributed objects and memory. This improves the overall stability of *HPX* and further simplifies writing real world applications.
- The minimal Boost version required to use *HPX* is now V1.49.0.
- This release coincides with the first release of HPXPI (V0.1.0), the first implementation of the XPI specification[5732].

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1086[5733] - Expose internal boost::shared_array to allow user management of array lifetime
- Issue #1083[5734] - Make shell examples copyable in docs
- Issue #1080[5735] - /threads{locality#*/total}/count/cumulative broken
- Issue #1079[5736] - Build problems on OS X
- Issue #1078[5737] - Improve robustness of component loading
- Issue #1077[5738] - Fix a missing enum definition for 'take' mode
- Issue #1076[5739] - Merge Jb master
- Issue #1075[5740] - Unknown CMake command "add_hpx_pseudo_target"
- Issue #1074[5741] - Implement `apply_continue_callback` and `apply_colocated_callback`
- Issue #1073[5742] - The new `apply_colocated` and `async_colocated` functions lead to automatic registered functions
- Issue #1071[5743] - Remove deferred_packaged_task
- Issue #1069[5744] - serialize_buffer with allocator fails at destruction
- Issue #1068[5745] - Coroutine include and forward declarations missing
- Issue #1067[5746] - Add allocator support to util::serialize_buffer

[5732] https://github.com/STEllAR-GROUP/hpxpi/blob/master/spec.pdf?raw=true
[5733] https://github.com/STEllAR-GROUP/hpx/issues/1086
[5734] https://github.com/STEllAR-GROUP/hpx/issues/1083
[5735] https://github.com/STEllAR-GROUP/hpx/issues/1080
[5736] https://github.com/STEllAR-GROUP/hpx/issues/1079
[5737] https://github.com/STEllAR-GROUP/hpx/issues/1078
[5738] https://github.com/STEllAR-GROUP/hpx/issues/1077
[5739] https://github.com/STEllAR-GROUP/hpx/issues/1076
[5740] https://github.com/STEllAR-GROUP/hpx/issues/1075
[5741] https://github.com/STEllAR-GROUP/hpx/issues/1074
[5742] https://github.com/STEllAR-GROUP/hpx/issues/1073
[5743] https://github.com/STEllAR-GROUP/hpx/issues/1071
[5744] https://github.com/STEllAR-GROUP/hpx/issues/1069
[5745] https://github.com/STEllAR-GROUP/hpx/issues/1068
[5746] https://github.com/STEllAR-GROUP/hpx/issues/1067

- Issue #1066[5747] - Allow for MPI_Init being called before HPX launches
- Issue #1065[5748] - AGAS cache isn't used/populated on worker localities
- Issue #1064[5749] - Reorder includes to ensure ws2 includes early
- Issue #1063[5750] - Add `hpx::runtime::suspend` and `hpx::runtime::resume`
- Issue #1062[5751] - Fix `async_continue` to properly handle return types
- Issue #1061[5752] - Implement `async_colocated` and `apply_colocated`
- Issue #1060[5753] - Implement minimal component migration
- Issue #1058[5754] - Remove `HPX_UTIL_TUPLE` from code base
- Issue #1057[5755] - Add performance counters for threading subsystem
- Issue #1055[5756] - Thread allocation uses two memory pools
- Issue #1053[5757] - Work stealing flawed
- Issue #1052[5758] - Fix a number of warnings
- Issue #1049[5759] - Fixes for TLS on OSX and more reliable test running
- Issue #1048[5760] - Fixing after 588 hang
- Issue #1047[5761] - Use port '0' for networking when using one locality
- Issue #1046[5762] - `composable_guard` test is broken when having more than one thread
- Issue #1045[5763] - Security missing headers
- Issue #1044[5764] - Native TLS on FreeBSD via __thread
- Issue #1043[5765] - `async` et.al. compute the wrong result type
- Issue #1042[5766] - `async` et.al. implicitly unwrap reference_wrappers
- Issue #1041[5767] - Remove redundant costly Kleene stars from regex searches
- Issue #1040[5768] - CMake script regex match patterns has unnecessary kleenes
- Issue #1039[5769] - Remove use of Boost.Move and replace with std::move and real rvalue refs

[5747] https://github.com/STEllAR-GROUP/hpx/issues/1066
[5748] https://github.com/STEllAR-GROUP/hpx/issues/1065
[5749] https://github.com/STEllAR-GROUP/hpx/issues/1064
[5750] https://github.com/STEllAR-GROUP/hpx/issues/1063
[5751] https://github.com/STEllAR-GROUP/hpx/issues/1062
[5752] https://github.com/STEllAR-GROUP/hpx/issues/1061
[5753] https://github.com/STEllAR-GROUP/hpx/issues/1060
[5754] https://github.com/STEllAR-GROUP/hpx/issues/1058
[5755] https://github.com/STEllAR-GROUP/hpx/issues/1057
[5756] https://github.com/STEllAR-GROUP/hpx/issues/1055
[5757] https://github.com/STEllAR-GROUP/hpx/issues/1053
[5758] https://github.com/STEllAR-GROUP/hpx/issues/1052
[5759] https://github.com/STEllAR-GROUP/hpx/issues/1049
[5760] https://github.com/STEllAR-GROUP/hpx/issues/1048
[5761] https://github.com/STEllAR-GROUP/hpx/issues/1047
[5762] https://github.com/STEllAR-GROUP/hpx/issues/1046
[5763] https://github.com/STEllAR-GROUP/hpx/issues/1045
[5764] https://github.com/STEllAR-GROUP/hpx/issues/1044
[5765] https://github.com/STEllAR-GROUP/hpx/issues/1043
[5766] https://github.com/STEllAR-GROUP/hpx/issues/1042
[5767] https://github.com/STEllAR-GROUP/hpx/issues/1041
[5768] https://github.com/STEllAR-GROUP/hpx/issues/1040
[5769] https://github.com/STEllAR-GROUP/hpx/issues/1039

- Issue #1038[5770] - Bump minimal required Boost to 1.49.0
- Issue #1037[5771] - Implicit unwrapping of futures in async broken
- Issue #1036[5772] - Scheduler hangs when user code attempts to "block" OS-threads
- Issue #1035[5773] - Idle-rate counter always reports 100% idle rate
- Issue #1034[5774] - Symbolic name registration causes application hangs
- Issue #1033[5775] - Application options read in from an options file generate an error message
- Issue #1032[5776] - `hpx::id_type` local reference counting is wrong
- Issue #1031[5777] - Negative entry in reference count table
- Issue #1030[5778] - Implement condition_variable
- Issue #1029[5779] - Deadlock in thread scheduling subsystem
- Issue #1028[5780] - HPX-thread cumulative count performance counters report incorrect value
- Issue #1027[5781] - Expose `hpx::thread_interrupted` error code as a separate exception type
- Issue #1026[5782] - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
- Issue #1025[5783] - `future::wait_for/wait_until` do not remove callback
- Issue #1024[5784] - Remove dependence to boost assert and create hpx assert
- Issue #1023[5785] - Segfaults with tcmalloc
- Issue #1022[5786] - prerequisites link in readme is broken
- Issue #1020[5787] - HPX Deadlock on external synchronization
- Issue #1019[5788] - Convert using `BOOST_ASSERT` to `HPX_ASSERT`
- Issue #1018[5789] - compiling bug with gcc 4.8.1
- Issue #1017[5790] - Possible crash in io_pool executor
- Issue #1016[5791] - Crash at startup
- Issue #1014[5792] - Implement Increment/Decrement Merging

5770 https://github.com/STEllAR-GROUP/hpx/issues/1038
5771 https://github.com/STEllAR-GROUP/hpx/issues/1037
5772 https://github.com/STEllAR-GROUP/hpx/issues/1036
5773 https://github.com/STEllAR-GROUP/hpx/issues/1035
5774 https://github.com/STEllAR-GROUP/hpx/issues/1034
5775 https://github.com/STEllAR-GROUP/hpx/issues/1033
5776 https://github.com/STEllAR-GROUP/hpx/issues/1032
5777 https://github.com/STEllAR-GROUP/hpx/issues/1031
5778 https://github.com/STEllAR-GROUP/hpx/issues/1030
5779 https://github.com/STEllAR-GROUP/hpx/issues/1029
5780 https://github.com/STEllAR-GROUP/hpx/issues/1028
5781 https://github.com/STEllAR-GROUP/hpx/issues/1027
5782 https://github.com/STEllAR-GROUP/hpx/issues/1026
5783 https://github.com/STEllAR-GROUP/hpx/issues/1025
5784 https://github.com/STEllAR-GROUP/hpx/issues/1024
5785 https://github.com/STEllAR-GROUP/hpx/issues/1023
5786 https://github.com/STEllAR-GROUP/hpx/issues/1022
5787 https://github.com/STEllAR-GROUP/hpx/issues/1020
5788 https://github.com/STEllAR-GROUP/hpx/issues/1019
5789 https://github.com/STEllAR-GROUP/hpx/issues/1018
5790 https://github.com/STEllAR-GROUP/hpx/issues/1017
5791 https://github.com/STEllAR-GROUP/hpx/issues/1016
5792 https://github.com/STEllAR-GROUP/hpx/issues/1014

- Issue #1013[5793] - Add more logging channels to enable greater control over logging granularity
- Issue #1012[5794] - `--hpx:debug-hpx-log` and `--hpx:debug-agas-log` lead to non-thread safe writes
- Issue #1011[5795] - After installation, running applications from the build/staging directory no longer works
- Issue #1010[5796] - Mergeable decrement requests are not being merged
- Issue #1009[5797] - `--hpx:list-symbolic-names` crashes
- Issue #1007[5798] - Components are not properly destroyed
- Issue #1006[5799] - Segfault/hang in set_data
- Issue #1003[5800] - Performance counter naming issue
- Issue #982[5801] - Race condition during startup
- Issue #912[5802] - OS X: component type not found in map
- Issue #663[5803] - Create a buildbot slave based on Clang 3.2/OSX
- Issue #636[5804] - Expose `this_locality::apply<act>(p1, p2);` for local execution
- Issue #197[5805] - Add `--console=address` option for PBS runs
- Issue #175[5806] - Asynchronous AGAS API

HPX V0.9.7 (Nov 13, 2013)

We have had over 1000 commits since the last release and we have closed over 180 tickets (bugs, feature requests, etc.).

General changes

- Ported HPX to BlueGene/Q
- Improved HPX support for Xeon/Phi accelerators
- Reimplemented `hpx::bind`, `hpx::tuple`, and `hpx::function` for better performance and better compliance with the C++11 Standard. Added `hpx::mem_fn`.
- Reworked `hpx::when_all` and `hpx::when_any` for better compliance with the ongoing C++ standardization effort, added heterogeneous version for those functions. Added `hpx::when_any_swapped`.
- Added `hpx::copy` as a precursor for a migrate functionality
- Added `hpx::get_ptr` allowing to directly access the memory underlying a given component
- Added the `hpx::lcos::broadcast`, `hpx::lcos::reduce`, and `hpx::lcos::fold` collective operations

[5793] https://github.com/STEllAR-GROUP/hpx/issues/1013
[5794] https://github.com/STEllAR-GROUP/hpx/issues/1012
[5795] https://github.com/STEllAR-GROUP/hpx/issues/1011
[5796] https://github.com/STEllAR-GROUP/hpx/issues/1010
[5797] https://github.com/STEllAR-GROUP/hpx/issues/1009
[5798] https://github.com/STEllAR-GROUP/hpx/issues/1007
[5799] https://github.com/STEllAR-GROUP/hpx/issues/1006
[5800] https://github.com/STEllAR-GROUP/hpx/issues/1003
[5801] https://github.com/STEllAR-GROUP/hpx/issues/982
[5802] https://github.com/STEllAR-GROUP/hpx/issues/912
[5803] https://github.com/STEllAR-GROUP/hpx/issues/663
[5804] https://github.com/STEllAR-GROUP/hpx/issues/636
[5805] https://github.com/STEllAR-GROUP/hpx/issues/197
[5806] https://github.com/STEllAR-GROUP/hpx/issues/175

- Added `hpx::get_locality_name` allowing to retrieve the name of any of the localities for the application.
- Added support for more flexible thread affinity control from the HPX command line, such as new modes for `--hpx:bind` (`balanced`, `scattered`, `compact`), improved default settings when running multiple localities on the same node.
- Added experimental executors for simpler thread pooling and scheduling. This API may change in the future as it will stay aligned with the ongoing C++ standardization efforts.
- Massively improved the performance of the HPX serialization code. Added partial support for zero copy serialization of array and bitwise-copyable types.
- General performance improvements of the code related to threads and futures.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1005[5807] - Allow one to disable array optimizations and zero copy optimizations for each parcelport
- Issue #1004[5808] - Generate new HPX logo image for the docs
- Issue #1002[5809] - If MPI parcelport is not available, running HPX under mpirun should fail
- Issue #1001[5810] - Zero copy serialization raises assert
- Issue #1000[5811] - Can't connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
- Issue #999[5812] - Optimize `hpx::when_n`
- Issue #998[5813] - Fixed const-correctness
- Issue #997[5814] - Making serialize_buffer::data() type save
- Issue #996[5815] - Memory leak in hpx::lcos::promise
- Issue #995[5816] - Race while registering pre-shutdown functions
- Issue #994[5817] - thread_rescheduling regression test does not compile
- Issue #992[5818] - Correct comments and messages
- Issue #991[5819] - setcap cap_sys_rawio=ep for power profiling causes an HPX application to abort
- Issue #989[5820] - Jacobi hangs during execution
- Issue #988[5821] - multiple_init test is failing

[5807] https://github.com/STEllAR-GROUP/hpx/issues/1005
[5808] https://github.com/STEllAR-GROUP/hpx/issues/1004
[5809] https://github.com/STEllAR-GROUP/hpx/issues/1002
[5810] https://github.com/STEllAR-GROUP/hpx/issues/1001
[5811] https://github.com/STEllAR-GROUP/hpx/issues/1000
[5812] https://github.com/STEllAR-GROUP/hpx/issues/999
[5813] https://github.com/STEllAR-GROUP/hpx/issues/998
[5814] https://github.com/STEllAR-GROUP/hpx/issues/997
[5815] https://github.com/STEllAR-GROUP/hpx/issues/996
[5816] https://github.com/STEllAR-GROUP/hpx/issues/995
[5817] https://github.com/STEllAR-GROUP/hpx/issues/994
[5818] https://github.com/STEllAR-GROUP/hpx/issues/992
[5819] https://github.com/STEllAR-GROUP/hpx/issues/991
[5820] https://github.com/STEllAR-GROUP/hpx/issues/989
[5821] https://github.com/STEllAR-GROUP/hpx/issues/988

- Issue #986[5822] - Can't call a function called "init" from "main" when using <hpx/hpx_main.hpp>
- Issue #984[5823] - Reference counting tests are failing
- Issue #983[5824] - thread_suspension_executor test fails
- Issue #980[5825] - Terminating HPX threads don't leave stack in virgin state
- Issue #979[5826] - Static scheduler not in documents
- Issue #978[5827] - Preprocessing limits are broken
- Issue #977[5828] - Make tests.regressions.lcos.future_hang_on_get shorter
- Issue #976[5829] - Wrong library order in pkgconfig
- Issue #975[5830] - Please reopen #963
- Issue #974[5831] - Option pu-offset ignored in fixing_588 branch
- Issue #972[5832] - Cannot use MKL with HPX
- Issue #969[5833] - Non-existent INI files requested on the command line via `--hpx:config` do not cause warnings or errors.
- Issue #968[5834] - Cannot build examples in fixing_588 branch
- Issue #967[5835] - Command line description of `--hpx:queuing` seems wrong
- Issue #966[5836] - `--hpx:print-bind` physical core numbers are wrong
- Issue #965[5837] - Deadlock when building in Release mode
- Issue #963[5838] - Not all worker threads are working
- Issue #962[5839] - Problem with SLURM integration
- Issue #961[5840] - `--hpx:print-bind` outputs incorrect information
- Issue #960[5841] - Fix cut and paste error in documentation of get_thread_priority
- Issue #959[5842] - Change link to boost.atomic in documentation to point to boost.org
- Issue #958[5843] - Undefined reference to intrusive_ptr_release
- Issue #957[5844] - Make tuple standard compliant

5822 https://github.com/STEllAR-GROUP/hpx/issues/986
5823 https://github.com/STEllAR-GROUP/hpx/issues/984
5824 https://github.com/STEllAR-GROUP/hpx/issues/983
5825 https://github.com/STEllAR-GROUP/hpx/issues/980
5826 https://github.com/STEllAR-GROUP/hpx/issues/979
5827 https://github.com/STEllAR-GROUP/hpx/issues/978
5828 https://github.com/STEllAR-GROUP/hpx/issues/977
5829 https://github.com/STEllAR-GROUP/hpx/issues/976
5830 https://github.com/STEllAR-GROUP/hpx/issues/975
5831 https://github.com/STEllAR-GROUP/hpx/issues/974
5832 https://github.com/STEllAR-GROUP/hpx/issues/972
5833 https://github.com/STEllAR-GROUP/hpx/issues/969
5834 https://github.com/STEllAR-GROUP/hpx/issues/968
5835 https://github.com/STEllAR-GROUP/hpx/issues/967
5836 https://github.com/STEllAR-GROUP/hpx/issues/966
5837 https://github.com/STEllAR-GROUP/hpx/issues/965
5838 https://github.com/STEllAR-GROUP/hpx/issues/963
5839 https://github.com/STEllAR-GROUP/hpx/issues/962
5840 https://github.com/STEllAR-GROUP/hpx/issues/961
5841 https://github.com/STEllAR-GROUP/hpx/issues/960
5842 https://github.com/STEllAR-GROUP/hpx/issues/959
5843 https://github.com/STEllAR-GROUP/hpx/issues/958
5844 https://github.com/STEllAR-GROUP/hpx/issues/957

- Issue #956[5845] - Segfault with a3382fb
- Issue #955[5846] - `--hpx:nodes` and `--hpx:nodefiles` do not work with foreign nodes
- Issue #954[5847] - Make order of arguments for hpx::async and hpx::broadcast consistent
- Issue #953[5848] - Cannot use MKL with HPX
- Issue #952[5849] - `register_[pre_]shutdown_function` never throw
- Issue #951[5850] - Assert when number of threads is greater than hardware concurrency
- Issue #948[5851] - `HPX_HAVE_GENERIC_CONTEXT_COROUTINES` conflicts with `HPX_HAVE_FIBER_BASED_COROUTINES`
- Issue #947[5852] - Need MPI_THREAD_MULTIPLE for backward compatibility
- Issue #946[5853] - HPX does not call `MPI_Finalize`
- Issue #945[5854] - Segfault with `hpx::lcos::broadcast`
- Issue #944[5855] - OS X: assertion `pu_offset_ < hardware_concurrency` failed
- Issue #943[5856] - #include <hpx/hpx_main.hpp> does not work
- Issue #942[5857] - Make the BG/Q work with -O3
- Issue #940[5858] - Use separator when concatenating locality name
- Issue #939[5859] - Refactor MPI parcelport to use `MPI_Wait` instead of multiple `MPI_Test` calls
- Issue #938[5860] - Want to officially access `client_base::gid_`
- Issue #937[5861] - `client_base::gid_` should be private``
- Issue #936[5862] - Want doxygen-like source code index
- Issue #935[5863] - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #933[5864] - Cannot build HPX with Boost 1.54.0
- Issue #932[5865] - Components are destructed too early
- Issue #931[5866] - Make HPX work on BG/Q
- Issue #930[5867] - make git-docs is broken

5845 https://github.com/STEllAR-GROUP/hpx/issues/956
5846 https://github.com/STEllAR-GROUP/hpx/issues/955
5847 https://github.com/STEllAR-GROUP/hpx/issues/954
5848 https://github.com/STEllAR-GROUP/hpx/issues/953
5849 https://github.com/STEllAR-GROUP/hpx/issues/952
5850 https://github.com/STEllAR-GROUP/hpx/issues/951
5851 https://github.com/STEllAR-GROUP/hpx/issues/948
5852 https://github.com/STEllAR-GROUP/hpx/issues/947
5853 https://github.com/STEllAR-GROUP/hpx/issues/946
5854 https://github.com/STEllAR-GROUP/hpx/issues/945
5855 https://github.com/STEllAR-GROUP/hpx/issues/944
5856 https://github.com/STEllAR-GROUP/hpx/issues/943
5857 https://github.com/STEllAR-GROUP/hpx/issues/942
5858 https://github.com/STEllAR-GROUP/hpx/issues/940
5859 https://github.com/STEllAR-GROUP/hpx/issues/939
5860 https://github.com/STEllAR-GROUP/hpx/issues/938
5861 https://github.com/STEllAR-GROUP/hpx/issues/937
5862 https://github.com/STEllAR-GROUP/hpx/issues/936
5863 https://github.com/STEllAR-GROUP/hpx/issues/935
5864 https://github.com/STEllAR-GROUP/hpx/issues/933
5865 https://github.com/STEllAR-GROUP/hpx/issues/932
5866 https://github.com/STEllAR-GROUP/hpx/issues/931
5867 https://github.com/STEllAR-GROUP/hpx/issues/930

- Issue #929[5868] - Generating index in docs broken
- Issue #928[5869] - Optimize `hpx::util::static_` for C++11 compilers supporting magic statics
- Issue #924[5870] - Make kill_process_tree (in process.py) more robust on Mac OSX
- Issue #923[5871] - Correct BLAS and RNPL cmake tests
- Issue #922[5872] - Cannot link against BLAS
- Issue #921[5873] - Implement `hpx::mem_fn`
- Issue #920[5874] - Output locality with `--hpx:print-bind`
- Issue #919[5875] - Correct grammar; simplify boolean expressions
- Issue #918[5876] - Link to hello_world.cpp is broken
- Issue #917[5877] - adapt cmake file to new boostbook version
- Issue #916[5878] - fix problem building documentation with xsltproc >= 1.1.27
- Issue #915[5879] - Add another TBBMalloc library search path
- Issue #914[5880] - Build problem with Intel compiler on Stampede (TACC)
- Issue #913[5881] - fix error messages in fibonacci examples
- Issue #911[5882] - Update OS X build instructions
- Issue #910[5883] - Want like to specify MPI_ROOT instead of compiler wrapper script
- Issue #909[5884] - Warning about void* arithmetic
- Issue #908[5885] - Buildbot for MIC is broken
- Issue #906[5886] - Can't use `--hpx:bind=balanced` with multiple MPI processes
- Issue #905[5887] - `--hpx:bind` documentation should describe full grammar
- Issue #904[5888] - Add hpx::lcos::fold and hpx::lcos::inverse_fold collective operation
- Issue #903[5889] - Add `hpx::when_any_swapped()`
- Issue #902[5890] - Add `hpx::lcos::reduce` collective operation

5868 https://github.com/STEllAR-GROUP/hpx/issues/929
5869 https://github.com/STEllAR-GROUP/hpx/issues/928
5870 https://github.com/STEllAR-GROUP/hpx/issues/924
5871 https://github.com/STEllAR-GROUP/hpx/issues/923
5872 https://github.com/STEllAR-GROUP/hpx/issues/922
5873 https://github.com/STEllAR-GROUP/hpx/issues/921
5874 https://github.com/STEllAR-GROUP/hpx/issues/920
5875 https://github.com/STEllAR-GROUP/hpx/issues/919
5876 https://github.com/STEllAR-GROUP/hpx/issues/918
5877 https://github.com/STEllAR-GROUP/hpx/issues/917
5878 https://github.com/STEllAR-GROUP/hpx/issues/916
5879 https://github.com/STEllAR-GROUP/hpx/issues/915
5880 https://github.com/STEllAR-GROUP/hpx/issues/914
5881 https://github.com/STEllAR-GROUP/hpx/issues/913
5882 https://github.com/STEllAR-GROUP/hpx/issues/911
5883 https://github.com/STEllAR-GROUP/hpx/issues/910
5884 https://github.com/STEllAR-GROUP/hpx/issues/909
5885 https://github.com/STEllAR-GROUP/hpx/issues/908
5886 https://github.com/STEllAR-GROUP/hpx/issues/906
5887 https://github.com/STEllAR-GROUP/hpx/issues/905
5888 https://github.com/STEllAR-GROUP/hpx/issues/904
5889 https://github.com/STEllAR-GROUP/hpx/issues/903
5890 https://github.com/STEllAR-GROUP/hpx/issues/902

- Issue #901[5891] - Web documentation is not searchable
- Issue #900[5892] - Web documentation for trunk has no index
- Issue #898[5893] - Some tests fail with GCC 4.8.1 and MPI parcel port
- Issue #897[5894] - HWLOC causes failures on Mac
- Issue #896[5895] - pu-offset leads to startup error
- Issue #895[5896] - `hpx::get_locality_name` not defined
- Issue #894[5897] - Race condition at shutdown
- Issue #893[5898] - `--hpx:print-bind` switches std::cout to hexadecimal mode
- Issue #892[5899] - `hwloc_topology_load` can be expensive – don't call multiple times
- Issue #891[5900] - The documentation for `get_locality_name` is wrong
- Issue #890[5901] - `--hpx:print-bind` should not exit
- Issue #889[5902] - `--hpx:debug-hpx-log=FILE` does not work
- Issue #888[5903] - MPI parcelport does not exit cleanly for –hpx:print-bind
- Issue #887[5904] - Choose thread affinities more cleverly
- Issue #886[5905] - Logging documentation is confusing
- Issue #885[5906] - Two threads are slower than one
- Issue #884[5907] - is_callable failing with member pointers in C++11
- Issue #883[5908] - Need help with is_callable_test
- Issue #882[5909] - tests.regressions.lcos.future_hang_on_get does not terminate
- Issue #881[5910] - tests/regressions/block_matrix/matrix.hh won't compile with GCC 4.8.1
- Issue #880[5911] - HPX does not work on OS X
- Issue #878[5912] - `future::unwrap` triggers assertion
- Issue #877[5913] - "make tests" has build errors on Ubuntu 12.10

[5891] https://github.com/STEllAR-GROUP/hpx/issues/901
[5892] https://github.com/STEllAR-GROUP/hpx/issues/900
[5893] https://github.com/STEllAR-GROUP/hpx/issues/898
[5894] https://github.com/STEllAR-GROUP/hpx/issues/897
[5895] https://github.com/STEllAR-GROUP/hpx/issues/896
[5896] https://github.com/STEllAR-GROUP/hpx/issues/895
[5897] https://github.com/STEllAR-GROUP/hpx/issues/894
[5898] https://github.com/STEllAR-GROUP/hpx/issues/893
[5899] https://github.com/STEllAR-GROUP/hpx/issues/892
[5900] https://github.com/STEllAR-GROUP/hpx/issues/891
[5901] https://github.com/STEllAR-GROUP/hpx/issues/890
[5902] https://github.com/STEllAR-GROUP/hpx/issues/889
[5903] https://github.com/STEllAR-GROUP/hpx/issues/888
[5904] https://github.com/STEllAR-GROUP/hpx/issues/887
[5905] https://github.com/STEllAR-GROUP/hpx/issues/886
[5906] https://github.com/STEllAR-GROUP/hpx/issues/885
[5907] https://github.com/STEllAR-GROUP/hpx/issues/884
[5908] https://github.com/STEllAR-GROUP/hpx/issues/883
[5909] https://github.com/STEllAR-GROUP/hpx/issues/882
[5910] https://github.com/STEllAR-GROUP/hpx/issues/881
[5911] https://github.com/STEllAR-GROUP/hpx/issues/880
[5912] https://github.com/STEllAR-GROUP/hpx/issues/878
[5913] https://github.com/STEllAR-GROUP/hpx/issues/877

- Issue #876[5914] - tcmalloc is used by default, even if it is not present
- Issue #875[5915] - global_fixture is defined in a header file
- Issue #874[5916] - Some tests take very long
- Issue #873[5917] - Add block-matrix code as regression test
- Issue #872[5918] - HPX documentation does not say how to run tests with detailed output
- Issue #871[5919] - All tests fail with "make test"
- Issue #870[5920] - Please explicitly disable serialization in classes that don't support it
- Issue #868[5921] - boost_any test failing
- Issue #867[5922] - Reduce the number of copies of `hpx::function` arguments
- Issue #863[5923] - Futures should not require a default constructor
- Issue #862[5924] - value_or_error shall not default construct its result
- Issue #861[5925] - `HPX_UNUSED` macro
- Issue #860[5926] - Add functionality to copy construct a component
- Issue #859[5927] - `hpx::endl` should flush
- Issue #858[5928] - Create `hpx::get_ptr`<> allowing to access component implementation
- Issue #855[5929] - Implement `hpx::INVOKE`
- Issue #854[5930] - `hpx/hpx.hpp` does not include `hpx/include/iostreams.hpp`
- Issue #853[5931] - Feature request: null future
- Issue #852[5932] - Feature request: Locality names
- Issue #851[5933] - `hpx::cout` output does not appear on screen
- Issue #849[5934] - All tests fail on OS X after installing
- Issue #848[5935] - Update OS X build instructions
- Issue #846[5936] - Update hpx_external_example

5914 https://github.com/STEllAR-GROUP/hpx/issues/876
5915 https://github.com/STEllAR-GROUP/hpx/issues/875
5916 https://github.com/STEllAR-GROUP/hpx/issues/874
5917 https://github.com/STEllAR-GROUP/hpx/issues/873
5918 https://github.com/STEllAR-GROUP/hpx/issues/872
5919 https://github.com/STEllAR-GROUP/hpx/issues/871
5920 https://github.com/STEllAR-GROUP/hpx/issues/870
5921 https://github.com/STEllAR-GROUP/hpx/issues/868
5922 https://github.com/STEllAR-GROUP/hpx/issues/867
5923 https://github.com/STEllAR-GROUP/hpx/issues/863
5924 https://github.com/STEllAR-GROUP/hpx/issues/862
5925 https://github.com/STEllAR-GROUP/hpx/issues/861
5926 https://github.com/STEllAR-GROUP/hpx/issues/860
5927 https://github.com/STEllAR-GROUP/hpx/issues/859
5928 https://github.com/STEllAR-GROUP/hpx/issues/858
5929 https://github.com/STEllAR-GROUP/hpx/issues/855
5930 https://github.com/STEllAR-GROUP/hpx/issues/854
5931 https://github.com/STEllAR-GROUP/hpx/issues/853
5932 https://github.com/STEllAR-GROUP/hpx/issues/852
5933 https://github.com/STEllAR-GROUP/hpx/issues/851
5934 https://github.com/STEllAR-GROUP/hpx/issues/849
5935 https://github.com/STEllAR-GROUP/hpx/issues/848
5936 https://github.com/STEllAR-GROUP/hpx/issues/846

- Issue #845[5937] - Issues with having both debug and release modules in the same directory
- Issue #844[5938] - Create configuration header
- Issue #843[5939] - Tests should use CTest
- Issue #842[5940] - Remove buffer_pool from MPI parcelport
- Issue #841[5941] - Add possibility to broadcast an index with hpx::lcos::broadcast
- Issue #838[5942] - Simplify `util::tuple`
- Issue #837[5943] - Adopt boost::tuple tests for `util::tuple`
- Issue #836[5944] - Adopt boost::function tests for `util::function`
- Issue #835[5945] - Tuple interface missing pieces
- Issue #833[5946] - Partially preprocessing files not working
- Issue #832[5947] - Native papi counters do not work with wild cards
- Issue #831[5948] - Arithmetics counter fails if only one parameter is given
- Issue #830[5949] - Convert hpx::util::function to use new scheme for serializing its base pointer
- Issue #829[5950] - Consistently use `decay<T>` instead of `remove_const< remove_reference<T>>`
- Issue #828[5951] - Update future implementation to N3721 and N3722
- Issue #827[5952] - Enable MPI parcelport for bootstrapping whenever application was started using mpirun
- Issue #826[5953] - Support command line option `--hpx:print-bind` even if `--hpx::bind` was not used
- Issue #825[5954] - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- Issue #824[5955] - Enable lambda functions to be used with hpx::async/hpx::apply
- Issue #823[5956] - Using a hashing filter
- Issue #822[5957] - Silence unused variable warning
- Issue #821[5958] - Detect if a function object is callable with given arguments
- Issue #820[5959] - Allow wildcards to be used for performance counter names

[5937] https://github.com/STEllAR-GROUP/hpx/issues/845
[5938] https://github.com/STEllAR-GROUP/hpx/issues/844
[5939] https://github.com/STEllAR-GROUP/hpx/issues/843
[5940] https://github.com/STEllAR-GROUP/hpx/issues/842
[5941] https://github.com/STEllAR-GROUP/hpx/issues/841
[5942] https://github.com/STEllAR-GROUP/hpx/issues/838
[5943] https://github.com/STEllAR-GROUP/hpx/issues/837
[5944] https://github.com/STEllAR-GROUP/hpx/issues/836
[5945] https://github.com/STEllAR-GROUP/hpx/issues/835
[5946] https://github.com/STEllAR-GROUP/hpx/issues/833
[5947] https://github.com/STEllAR-GROUP/hpx/issues/832
[5948] https://github.com/STEllAR-GROUP/hpx/issues/831
[5949] https://github.com/STEllAR-GROUP/hpx/issues/830
[5950] https://github.com/STEllAR-GROUP/hpx/issues/829
[5951] https://github.com/STEllAR-GROUP/hpx/issues/828
[5952] https://github.com/STEllAR-GROUP/hpx/issues/827
[5953] https://github.com/STEllAR-GROUP/hpx/issues/826
[5954] https://github.com/STEllAR-GROUP/hpx/issues/825
[5955] https://github.com/STEllAR-GROUP/hpx/issues/824
[5956] https://github.com/STEllAR-GROUP/hpx/issues/823
[5957] https://github.com/STEllAR-GROUP/hpx/issues/822
[5958] https://github.com/STEllAR-GROUP/hpx/issues/821
[5959] https://github.com/STEllAR-GROUP/hpx/issues/820

- Issue #819[5960] - Make the AGAS symbolic name registry distributed
- Issue #818[5961] - Add future::then() overload taking an executor
- Issue #817[5962] - Fixed typo
- Issue #815[5963] - Create an lco that is performing an efficient broadcast of actions
- Issue #814[5964] - Papi counters cannot specify thread#* to get the counts for all threads
- Issue #813[5965] - Scoped unlock
- Issue #811[5966] - simple_central_tuplespace_client run error
- Issue #810[5967] - ostream error when << any objects
- Issue #809[5968] - Optimize parcel serialization
- Issue #808[5969] - HPX applications throw exception when executed from the build directory
- Issue #807[5970] - Create performance counters exposing overall AGAS statistics
- Issue #795[5971] - Create timed make_ready_future
- Issue #794[5972] - Create heterogeneous `when_all`/`when_any`/etc.
- Issue #721[5973] - Make HPX usable for Xeon Phi
- Issue #694[5974] - CMake should complain if you attempt to build an example without its dependencies
- Issue #692[5975] - SLURM support broken
- Issue #683[5976] - python/hpx/process.py imports epoll on all platforms
- Issue #619[5977] - Automate the doc building process
- Issue #600[5978] - GTC performance broken
- Issue #577[5979] - Allow for zero copy serialization/networking
- Issue #551[5980] - Change executable names to have debug postfix in Debug builds
- Issue #544[5981] - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init
- Issue #534[5982] - `hpx::init` should take functions by `std::function` and should accept all forms of hpx_main

5960 https://github.com/STEllAR-GROUP/hpx/issues/819
5961 https://github.com/STEllAR-GROUP/hpx/issues/818
5962 https://github.com/STEllAR-GROUP/hpx/issues/817
5963 https://github.com/STEllAR-GROUP/hpx/issues/815
5964 https://github.com/STEllAR-GROUP/hpx/issues/814
5965 https://github.com/STEllAR-GROUP/hpx/issues/813
5966 https://github.com/STEllAR-GROUP/hpx/issues/811
5967 https://github.com/STEllAR-GROUP/hpx/issues/810
5968 https://github.com/STEllAR-GROUP/hpx/issues/809
5969 https://github.com/STEllAR-GROUP/hpx/issues/808
5970 https://github.com/STEllAR-GROUP/hpx/issues/807
5971 https://github.com/STEllAR-GROUP/hpx/issues/795
5972 https://github.com/STEllAR-GROUP/hpx/issues/794
5973 https://github.com/STEllAR-GROUP/hpx/issues/721
5974 https://github.com/STEllAR-GROUP/hpx/issues/694
5975 https://github.com/STEllAR-GROUP/hpx/issues/692
5976 https://github.com/STEllAR-GROUP/hpx/issues/683
5977 https://github.com/STEllAR-GROUP/hpx/issues/619
5978 https://github.com/STEllAR-GROUP/hpx/issues/600
5979 https://github.com/STEllAR-GROUP/hpx/issues/577
5980 https://github.com/STEllAR-GROUP/hpx/issues/551
5981 https://github.com/STEllAR-GROUP/hpx/issues/544
5982 https://github.com/STEllAR-GROUP/hpx/issues/534

- Issue #508[5983] - FindPackage fails to set FOO_LIBRARY_DIR
- Issue #506[5984] - Add cmake support to generate ini files for external applications
- Issue #470[5985] - Changing build-type after configure does not update boost library names
- Issue #453[5986] - Document `hpx_run_tests.py`
- Issue #445[5987] - Significant performance mismatch between MPI and HPX in SMP for allgather example
- Issue #443[5988] - Make docs viewable from build directory
- Issue #421[5989] - Support multiple HPX instances per node in a batch environment like PBS or SLURM
- Issue #316[5990] - Add message size limitation
- Issue #249[5991] - Clean up locking code in big boot barrier
- Issue #136[5992] - Persistent CMake variables need to be marked as cache variables

HPX V0.9.6 (Jul 30, 2013)

We have had over 1200 commits since the last release and we have closed roughly 140 tickets (bugs, feature requests, etc.).

General changes

The major new features in this release are:

- We further consolidated the API exposed by *HPX*. We aligned our APIs as much as possible with the existing C++11 Standard[5993] and related proposals to the C++ standardization committee (such as N3632[5994] and N3857[5995]).
- We implemented a first version of a distributed AGAS service which essentially eliminates all explicit AGAS network traffic.
- We created a native ibverbs parcelport allowing to take advantage of the superior latency and bandwidth characteristics of Infiniband networks.
- We successfully ported *HPX* to the Xeon Phi platform.
- Support for the SLURM scheduling system was implemented.
- Major efforts have been dedicated to improving the performance counter framework, numerous new counters were implemented and new APIs were added.
- We added a modular parcel compression system allowing to improve bandwidth utilization (by reducing the overall size of the transferred data).

[5983] https://github.com/STEllAR-GROUP/hpx/issues/508
[5984] https://github.com/STEllAR-GROUP/hpx/issues/506
[5985] https://github.com/STEllAR-GROUP/hpx/issues/470
[5986] https://github.com/STEllAR-GROUP/hpx/issues/453
[5987] https://github.com/STEllAR-GROUP/hpx/issues/445
[5988] https://github.com/STEllAR-GROUP/hpx/issues/443
[5989] https://github.com/STEllAR-GROUP/hpx/issues/421
[5990] https://github.com/STEllAR-GROUP/hpx/issues/316
[5991] https://github.com/STEllAR-GROUP/hpx/issues/249
[5992] https://github.com/STEllAR-GROUP/hpx/issues/136
[5993] http://www.open-std.org/jtc1/sc22/wg21
[5994] http://wg21.link/n3632
[5995] http://wg21.link/n3857

- We added a modular parcel coalescing system allowing to combine several parcels into larger messages. This reduces latencies introduced by the communication layer.
- Added an experimental executors API allowing to use different scheduling policies for different parts of the code. This API has been modelled after the Standards proposal N3562[5996]. This API is bound to change in the future, though.
- Added minimal security support for localities which is enforced on the parcelport level. This support is preliminary and experimental and might change in the future.
- We created a parcelport using low level MPI functions. This is in support of legacy applications which are to be gradually ported and to support platforms where MPI is the only available portable networking layer.
- We added a preliminary and experimental implementation of a tuple-space object which exposes an interface similar to such systems described in the literature (see for instance The Linda Coordination Language[5997]).

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release. This is again a very long list of newly implemented features and fixed issues.

- Issue #806[5998] - make (all) in examples folder does nothing
- Issue #805[5999] - Adding the introduction and fixing DOCBOOK dependencies for Windows use
- Issue #804[6000] - Add stackless (non-suspendable) thread type
- Issue #803[6001] - Create proper serialization support functions for util::tuple
- Issue #800[6002] - Add possibility to disable array optimizations during serialization
- Issue #798[6003] - HPX_LIMIT does not work for local dataflow
- Issue #797[6004] - Create a parcelport which uses MPI
- Issue #796[6005] - Problem with Large Numbers of Threads
- Issue #793[6006] - Changing dataflow test case to hang consistently
- Issue #792[6007] - CMake Error
- Issue #791[6008] - Problems with local::dataflow
- Issue #790[6009] - wait_for() doesn't compile
- Issue #789[6010] - HPX with Intel compiler segfaults
- Issue #788[6011] - Intel compiler support

[5996] http://wg21.link/n3562
[5997] https://en.wikipedia.org/wiki/Linda_(coordination_language)
[5998] https://github.com/STEllAR-GROUP/hpx/issues/806
[5999] https://github.com/STEllAR-GROUP/hpx/issues/805
[6000] https://github.com/STEllAR-GROUP/hpx/issues/804
[6001] https://github.com/STEllAR-GROUP/hpx/issues/803
[6002] https://github.com/STEllAR-GROUP/hpx/issues/800
[6003] https://github.com/STEllAR-GROUP/hpx/issues/798
[6004] https://github.com/STEllAR-GROUP/hpx/issues/797
[6005] https://github.com/STEllAR-GROUP/hpx/issues/796
[6006] https://github.com/STEllAR-GROUP/hpx/issues/793
[6007] https://github.com/STEllAR-GROUP/hpx/issues/792
[6008] https://github.com/STEllAR-GROUP/hpx/issues/791
[6009] https://github.com/STEllAR-GROUP/hpx/issues/790
[6010] https://github.com/STEllAR-GROUP/hpx/issues/789
[6011] https://github.com/STEllAR-GROUP/hpx/issues/788

- Issue #787[6012] - Fixed SFINAEd specializations
- Issue #786[6013] - Memory issues during benchmarking.
- Issue #785[6014] - Create an API allowing to register external threads with HPX
- Issue #784[6015] - util::plugin is throwing an error when a symbol is not found
- Issue #783[6016] - How does hpx:bind work?
- Issue #782[6017] - Added quotes around STRING REPLACE potentially empty arguments
- Issue #781[6018] - Make sure no exceptions propagate into the thread manager
- Issue #780[6019] - Allow arithmetics performance counters to expand its parameters
- Issue #779[6020] - Test case for 778
- Issue #778[6021] - Swapping futures segfaults
- Issue #777[6022] - hpx::lcos::details::when_xxx don't restore completion handlers
- Issue #776[6023] - Compiler chokes on dataflow overload with launch policy
- Issue #775[6024] - Runtime error with local dataflow (copying futures?)
- Issue #774[6025] - Using local dataflow without explicit namespace
- Issue #773[6026] - Local dataflow with unwrap: functor operators need to be const
- Issue #772[6027] - Allow (remote) actions to return a future
- Issue #771[6028] - Setting HPX_LIMIT gives huge boost MPL errors
- Issue #770[6029] - Add launch policy to (local) dataflow
- Issue #769[6030] - Make compile time configuration information available
- Issue #768[6031] - Const correctness problem in local dataflow
- Issue #767[6032] - Add launch policies to async
- Issue #766[6033] - Mark data structures for optimized (array based) serialization
- Issue #765[6034] - Align hpx::any with N3508: Any Library Proposal (Revision 2)

6012 https://github.com/STEllAR-GROUP/hpx/issues/787
6013 https://github.com/STEllAR-GROUP/hpx/issues/786
6014 https://github.com/STEllAR-GROUP/hpx/issues/785
6015 https://github.com/STEllAR-GROUP/hpx/issues/784
6016 https://github.com/STEllAR-GROUP/hpx/issues/783
6017 https://github.com/STEllAR-GROUP/hpx/issues/782
6018 https://github.com/STEllAR-GROUP/hpx/issues/781
6019 https://github.com/STEllAR-GROUP/hpx/issues/780
6020 https://github.com/STEllAR-GROUP/hpx/issues/779
6021 https://github.com/STEllAR-GROUP/hpx/issues/778
6022 https://github.com/STEllAR-GROUP/hpx/issues/777
6023 https://github.com/STEllAR-GROUP/hpx/issues/776
6024 https://github.com/STEllAR-GROUP/hpx/issues/775
6025 https://github.com/STEllAR-GROUP/hpx/issues/774
6026 https://github.com/STEllAR-GROUP/hpx/issues/773
6027 https://github.com/STEllAR-GROUP/hpx/issues/772
6028 https://github.com/STEllAR-GROUP/hpx/issues/771
6029 https://github.com/STEllAR-GROUP/hpx/issues/770
6030 https://github.com/STEllAR-GROUP/hpx/issues/769
6031 https://github.com/STEllAR-GROUP/hpx/issues/768
6032 https://github.com/STEllAR-GROUP/hpx/issues/767
6033 https://github.com/STEllAR-GROUP/hpx/issues/766
6034 https://github.com/STEllAR-GROUP/hpx/issues/765

- Issue #764[6035] - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations
- Issue #762[6036] - added a human readable output for the ping pong example
- Issue #761[6037] - Ambiguous typename when constructing derived component
- Issue #760[6038] - Simple components can not be derived
- Issue #759[6039] - make install doesn't give a complete install
- Issue #758[6040] - Stack overflow when using locking_hook<>
- Issue #757[6041] - copy paste error; unsupported function overloading
- Issue #756[6042] - GTCX runtime issue in Gordon
- Issue #755[6043] - Papi counters don't work with reset and evaluate API's
- Issue #753[6044] - cmake bugfix and improved component action docs
- Issue #752[6045] - hpx simple component docs
- Issue #750[6046] - Add hpx::util::any
- Issue #749[6047] - Thread phase counter is not reset
- Issue #748[6048] - Memory performance counter are not registered
- Issue #747[6049] - Create performance counters exposing arithmetic operations
- Issue #745[6050] - apply_callback needs to invoke callback when applied locally
- Issue #744[6051] - CMake fixes
- Issue #743[6052] - Problem Building github version of HPX
- Issue #742[6053] - Remove HPX_STD_BIND
- Issue #741[6054] - assertion 'px != 0' failed: HPX(assertion_failure) for low numbers of OS threads
- Issue #739[6055] - Performance counters do not count to the end of the program or evaluation
- Issue #738[6056] - Dedicated AGAS server runs don't work; console ignores -a option.
- Issue #737[6057] - Missing bind overloads

6035 https://github.com/STEllAR-GROUP/hpx/issues/764
6036 https://github.com/STEllAR-GROUP/hpx/issues/762
6037 https://github.com/STEllAR-GROUP/hpx/issues/761
6038 https://github.com/STEllAR-GROUP/hpx/issues/760
6039 https://github.com/STEllAR-GROUP/hpx/issues/759
6040 https://github.com/STEllAR-GROUP/hpx/issues/758
6041 https://github.com/STEllAR-GROUP/hpx/issues/757
6042 https://github.com/STEllAR-GROUP/hpx/issues/756
6043 https://github.com/STEllAR-GROUP/hpx/issues/755
6044 https://github.com/STEllAR-GROUP/hpx/issues/753
6045 https://github.com/STEllAR-GROUP/hpx/issues/752
6046 https://github.com/STEllAR-GROUP/hpx/issues/750
6047 https://github.com/STEllAR-GROUP/hpx/issues/749
6048 https://github.com/STEllAR-GROUP/hpx/issues/748
6049 https://github.com/STEllAR-GROUP/hpx/issues/747
6050 https://github.com/STEllAR-GROUP/hpx/issues/745
6051 https://github.com/STEllAR-GROUP/hpx/issues/744
6052 https://github.com/STEllAR-GROUP/hpx/issues/743
6053 https://github.com/STEllAR-GROUP/hpx/issues/742
6054 https://github.com/STEllAR-GROUP/hpx/issues/741
6055 https://github.com/STEllAR-GROUP/hpx/issues/739
6056 https://github.com/STEllAR-GROUP/hpx/issues/738
6057 https://github.com/STEllAR-GROUP/hpx/issues/737

- Issue #736[6058] - Performance counter wildcards do not always work
- Issue #735[6059] - Create native ibverbs parcelport based on rdma operations
- Issue #734[6060] - Threads stolen performance counter total is incorrect
- Issue #733[6061] - Test benchmarks need to be checked and fixed
- Issue #732[6062] - Build fails with Mac, using mac ports clang-3.3 on latest git branch
- Issue #731[6063] - Add global start/stop API for performance counters
- Issue #730[6064] - Performance counter values are apparently incorrect
- Issue #729[6065] - Unhandled switch
- Issue #728[6066] - Serialization of hpx::util::function between two localities causes seg faults
- Issue #727[6067] - Memory counters on Mac OS X
- Issue #725[6068] - Restore original thread priority on resume
- Issue #724[6069] - Performance benchmarks do not depend on main HPX libraries
- Issue #723[6070] - [teletype]–hpx:nodes=``cat $PBS_NODEFILE`` works; –hpx:nodefile=$PBS_NODEFILE does not.[c++]
- Issue #722[6071] - Fix binding const member functions as actions
- Issue #719[6072] - Create performance counter exposing compression ratio
- Issue #718[6073] - Add possibility to compress parcel data
- Issue #717[6074] - strip_credit_from_gid has misleading semantics
- Issue #716[6075] - Non-option arguments to programs run using `pbsdsh` must be before `--hpx:nodes`, contrary to directions
- Issue #715[6076] - Re-thrown exceptions should retain the original call site
- Issue #714[6077] - failed assertion in debug mode
- Issue #713[6078] - Add performance counters monitoring connection caches
- Issue #712[6079] - Adjust parcel related performance counters to be connection type specific

[6058] https://github.com/STEllAR-GROUP/hpx/issues/736
[6059] https://github.com/STEllAR-GROUP/hpx/issues/735
[6060] https://github.com/STEllAR-GROUP/hpx/issues/734
[6061] https://github.com/STEllAR-GROUP/hpx/issues/733
[6062] https://github.com/STEllAR-GROUP/hpx/issues/732
[6063] https://github.com/STEllAR-GROUP/hpx/issues/731
[6064] https://github.com/STEllAR-GROUP/hpx/issues/730
[6065] https://github.com/STEllAR-GROUP/hpx/issues/729
[6066] https://github.com/STEllAR-GROUP/hpx/issues/728
[6067] https://github.com/STEllAR-GROUP/hpx/issues/727
[6068] https://github.com/STEllAR-GROUP/hpx/issues/725
[6069] https://github.com/STEllAR-GROUP/hpx/issues/724
[6070] https://github.com/STEllAR-GROUP/hpx/issues/723
[6071] https://github.com/STEllAR-GROUP/hpx/issues/722
[6072] https://github.com/STEllAR-GROUP/hpx/issues/719
[6073] https://github.com/STEllAR-GROUP/hpx/issues/718
[6074] https://github.com/STEllAR-GROUP/hpx/issues/717
[6075] https://github.com/STEllAR-GROUP/hpx/issues/716
[6076] https://github.com/STEllAR-GROUP/hpx/issues/715
[6077] https://github.com/STEllAR-GROUP/hpx/issues/714
[6078] https://github.com/STEllAR-GROUP/hpx/issues/713
[6079] https://github.com/STEllAR-GROUP/hpx/issues/712

- Issue #711[6080] - configuration failure
- Issue #710[6081] - Error "timed out while trying to find room in the connection cache" when trying to start multiple localities on a single computer
- Issue #709[6082] - Add new thread state 'staged' referring to task descriptions
- Issue #708[6083] - Detect/mitigate bad non-system installs of GCC on Redhat systems
- Issue #707[6084] - Many examples do not link with Git HEAD version
- Issue #706[6085] - `hpx::init` removes portions of non-option command line arguments before last = sign
- Issue #705[6086] - Create rolling average and median aggregating performance counters
- Issue #704[6087] - Create performance counter to expose thread queue waiting time
- Issue #703[6088] - Add support to HPX build system to find librcrtool.a and related headers
- Issue #699[6089] - Generalize instrumentation support
- Issue #698[6090] - compilation failure with hwloc absent
- Issue #697[6091] - Performance counter counts should be zero indexed
- Issue #696[6092] - Distributed problem
- Issue #695[6093] - Bad perf counter time printed
- Issue #693[6094] - `--help` doesn't print component specific command line options
- Issue #692[6095] - SLURM support broken
- Issue #691[6096] - exception while executing any application linked with hwloc
- Issue #690[6097] - thread_id_test and thread_launcher_test failing
- Issue #689[6098] - Make the buildbots use hwloc
- Issue #687[6099] - compilation error fix (hwloc_topology)
- Issue #686[6100] - Linker Error for Applications
- Issue #684[6101] - Pinning of service thread fails when number of worker threads equals the number of cores
- Issue #682[6102] - Add performance counters exposing number of stolen threads

[6080] https://github.com/STEllAR-GROUP/hpx/issues/711
[6081] https://github.com/STEllAR-GROUP/hpx/issues/710
[6082] https://github.com/STEllAR-GROUP/hpx/issues/709
[6083] https://github.com/STEllAR-GROUP/hpx/issues/708
[6084] https://github.com/STEllAR-GROUP/hpx/issues/707
[6085] https://github.com/STEllAR-GROUP/hpx/issues/706
[6086] https://github.com/STEllAR-GROUP/hpx/issues/705
[6087] https://github.com/STEllAR-GROUP/hpx/issues/704
[6088] https://github.com/STEllAR-GROUP/hpx/issues/703
[6089] https://github.com/STEllAR-GROUP/hpx/issues/699
[6090] https://github.com/STEllAR-GROUP/hpx/issues/698
[6091] https://github.com/STEllAR-GROUP/hpx/issues/697
[6092] https://github.com/STEllAR-GROUP/hpx/issues/696
[6093] https://github.com/STEllAR-GROUP/hpx/issues/695
[6094] https://github.com/STEllAR-GROUP/hpx/issues/693
[6095] https://github.com/STEllAR-GROUP/hpx/issues/692
[6096] https://github.com/STEllAR-GROUP/hpx/issues/691
[6097] https://github.com/STEllAR-GROUP/hpx/issues/690
[6098] https://github.com/STEllAR-GROUP/hpx/issues/689
[6099] https://github.com/STEllAR-GROUP/hpx/issues/687
[6100] https://github.com/STEllAR-GROUP/hpx/issues/686
[6101] https://github.com/STEllAR-GROUP/hpx/issues/684
[6102] https://github.com/STEllAR-GROUP/hpx/issues/682

- Issue #681[6103] - Add apply_continue for asynchronous chaining of actions
- Issue #679[6104] - Remove obsolete async_callback API functions
- Issue #678[6105] - Add new API for setting/triggering LCOs
- Issue #677[6106] - Add async_continue for true continuation style actions
- Issue #676[6107] - Buildbot for gcc 4.4 broken
- Issue #675[6108] - Partial preprocessing broken
- Issue #674[6109] - HPX segfaults when built with gcc 4.7
- Issue #673[6110] - `use_guard_pages` has inconsistent preprocessor guards
- Issue #672[6111] - External build breaks if library path has spaces
- Issue #671[6112] - release tarballs are tarbombs
- Issue #670[6113] - CMake won't find Boost headers in layout=versioned install
- Issue #669[6114] - Links in docs to source files broken if not installed
- Issue #667[6115] - Not reading ini file properly
- Issue #664[6116] - Adapt new meanings of 'const' and 'mutable'
- Issue #661[6117] - Implement BTL Parcel port
- Issue #655[6118] - Make HPX work with the "decltype" result_of
- Issue #647[6119] - documentation for specifying the number of high priority threads `--hpx:high-priority-threads`
- Issue #643[6120] - Error parsing host file
- Issue #642[6121] - HWLoc issue with TAU
- Issue #639[6122] - Logging potentially suspends a running thread
- Issue #634[6123] - Improve error reporting from parcel layer
- Issue #627[6124] - Add tests for async and apply overloads that accept regular C++ functions
- Issue #626[6125] - hpx/future.hpp header

6103 https://github.com/STEllAR-GROUP/hpx/issues/681
6104 https://github.com/STEllAR-GROUP/hpx/issues/679
6105 https://github.com/STEllAR-GROUP/hpx/issues/678
6106 https://github.com/STEllAR-GROUP/hpx/issues/677
6107 https://github.com/STEllAR-GROUP/hpx/issues/676
6108 https://github.com/STEllAR-GROUP/hpx/issues/675
6109 https://github.com/STEllAR-GROUP/hpx/issues/674
6110 https://github.com/STEllAR-GROUP/hpx/issues/673
6111 https://github.com/STEllAR-GROUP/hpx/issues/672
6112 https://github.com/STEllAR-GROUP/hpx/issues/671
6113 https://github.com/STEllAR-GROUP/hpx/issues/670
6114 https://github.com/STEllAR-GROUP/hpx/issues/669
6115 https://github.com/STEllAR-GROUP/hpx/issues/667
6116 https://github.com/STEllAR-GROUP/hpx/issues/664
6117 https://github.com/STEllAR-GROUP/hpx/issues/661
6118 https://github.com/STEllAR-GROUP/hpx/issues/655
6119 https://github.com/STEllAR-GROUP/hpx/issues/647
6120 https://github.com/STEllAR-GROUP/hpx/issues/643
6121 https://github.com/STEllAR-GROUP/hpx/issues/642
6122 https://github.com/STEllAR-GROUP/hpx/issues/639
6123 https://github.com/STEllAR-GROUP/hpx/issues/634
6124 https://github.com/STEllAR-GROUP/hpx/issues/627
6125 https://github.com/STEllAR-GROUP/hpx/issues/626

- Issue #601[6126] - Intel support
- Issue #557[6127] - Remove action codes
- Issue #531[6128] - AGAS request and response classes should use switch statements
- Issue #529[6129] - Investigate the state of hwloc support
- Issue #526[6130] - Make HPX aware of hyper-threading
- Issue #518[6131] - Create facilities allowing to use plain arrays as action arguments
- Issue #473[6132] - hwloc thread binding is broken on CPUs with hyperthreading
- Issue #383[6133] - Change result type detection for hpx::util::bind to use result_of protocol
- Issue #341[6134] - Consolidate route code
- Issue #219[6135] - Only copy arguments into actions once
- Issue #177[6136] - Implement distributed AGAS
- Issue #43[6137] - Support for Darwin (Xcode + Clang)

HPX V0.9.5 (Jan 16, 2013)

We have had over 1000 commits since the last release and we have closed roughly 150 tickets (bugs, feature requests, etc.).

General changes

This release is continuing along the lines of code and API consolidation, and overall usability inprovements. We dedicated much attention to performance and we were able to significantly improve the threading and networking subsystems.

We successfully ported *HPX* to the Android platform. *HPX* applications now not only can run on mobile devices, but we support heterogeneous applications running across architecture boundaries. At the Supercomputing Conference 2012 we demonstrated connecting Android tablets to simulations running on a Linux cluster. The Android tablet was used to query performance counters from the Linux simulation and to steer its parameters.

We successfully ported *HPX* to Mac OSX (using the Clang compiler). Thanks to Pyry Jahkola for contributing the corresponding patches. Please see the section macos_installation for more details.

We made a special effort to make HPX usable in highly concurrent use cases. Many of the HPX API functions which possibly take longer than 100 microseconds to execute now can be invoked asynchronously. We added uniform support for composing futures which simplifies to write asynchronous code. HPX actions (function objects encapsulating possibly concurrent remote function invocations) are now well integrated with all other API facilities such like `hpx::bind`.

[6126] https://github.com/STEllAR-GROUP/hpx/issues/601
[6127] https://github.com/STEllAR-GROUP/hpx/issues/557
[6128] https://github.com/STEllAR-GROUP/hpx/issues/531
[6129] https://github.com/STEllAR-GROUP/hpx/issues/529
[6130] https://github.com/STEllAR-GROUP/hpx/issues/526
[6131] https://github.com/STEllAR-GROUP/hpx/issues/518
[6132] https://github.com/STEllAR-GROUP/hpx/issues/473
[6133] https://github.com/STEllAR-GROUP/hpx/issues/383
[6134] https://github.com/STEllAR-GROUP/hpx/issues/341
[6135] https://github.com/STEllAR-GROUP/hpx/issues/219
[6136] https://github.com/STEllAR-GROUP/hpx/issues/177
[6137] https://github.com/STEllAR-GROUP/hpx/issues/43

All of the API has been aligned as much as possible with established paradigms. HPX now mirrors many of the facilities as defined in the C++11 Standard, such as `hpx::thread`, `hpx::function`, `hpx::future`, etc.

A lot of work has been put into improving the documentation. Many of the API functions are documented now, concepts are explained in detail, and examples are better described than before. The new documentation index enables finding information with lesser effort.

This is the first release of HPX we perform after the move to Github[6138] This step has enabled a wider participation from the community and further encourages us in our decision to release HPX as a true open source library (HPX is licensed under the very liberal Boost Software License[6139]).

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release. This is by far the longest list of newly implemented features and fixed issues for any of HPX' releases so far.

- Issue #666[6140] - Segfault on calling hpx::finalize twice
- Issue #665[6141] - Adding declaration num_of_cores
- Issue #662[6142] - pkgconfig is building wrong
- Issue #660[6143] - Need uninterrupt function
- Issue #659[6144] - Move our logging library into a different namespace
- Issue #658[6145] - Dynamic performance counter types are broken
- Issue #657[6146] - HPX v0.9.5 (RC1) hello_world example segfaulting
- Issue #656[6147] - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- Issue #654[6148] - Integrate the Boost auto_index tool with documentation
- Issue #653[6149] - Make HPX build on OS X + Clang + libc++
- Issue #651[6150] - Add fine-grained control for thread pinning
- Issue #650[6151] - Command line no error message when using -hpx:(anything)
- Issue #645[6152] - Command line aliases don't work in [teletype]``@file``[c++]
- Issue #644[6153] - Terminated threads are not always properly cleaned up
- Issue #640[6154] - `future_data<T>::set_on_completed_` used without locks
- Issue #638[6155] - hpx build with intel compilers fails on linux

6138 https://github.com/STEllAR-GROUP/hpx/
6139 https://www.boost.org/LICENSE_1_0.txt
6140 https://github.com/STEllAR-GROUP/hpx/issues/666
6141 https://github.com/STEllAR-GROUP/hpx/issues/665
6142 https://github.com/STEllAR-GROUP/hpx/issues/662
6143 https://github.com/STEllAR-GROUP/hpx/issues/660
6144 https://github.com/STEllAR-GROUP/hpx/issues/659
6145 https://github.com/STEllAR-GROUP/hpx/issues/658
6146 https://github.com/STEllAR-GROUP/hpx/issues/657
6147 https://github.com/STEllAR-GROUP/hpx/issues/656
6148 https://github.com/STEllAR-GROUP/hpx/issues/654
6149 https://github.com/STEllAR-GROUP/hpx/issues/653
6150 https://github.com/STEllAR-GROUP/hpx/issues/651
6151 https://github.com/STEllAR-GROUP/hpx/issues/650
6152 https://github.com/STEllAR-GROUP/hpx/issues/645
6153 https://github.com/STEllAR-GROUP/hpx/issues/644
6154 https://github.com/STEllAR-GROUP/hpx/issues/640
6155 https://github.com/STEllAR-GROUP/hpx/issues/638

- Issue #637[6156] - –copy-dt-needed-entries breaks with gold
- Issue #635[6157] - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- Issue #633[6158] - Re-add examples to final 0.9.5 release
- Issue #632[6159] - Example `thread_aware_timer` is broken
- Issue #631[6160] - FFT application throws error in parcellayer
- Issue #630[6161] - Event synchronization example is broken
- Issue #629[6162] - Waiting on futures hangs
- Issue #628[6163] - Add an `HPX_ALWAYS_ASSERT` macro
- Issue #625[6164] - Port coroutines context switch benchmark
- Issue #621[6165] - New INI section for stack sizes
- Issue #618[6166] - pkg_config support does not work with a HPX debug build
- Issue #617[6167] - hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: 'get_thread_id' was not declared in this scope
- Issue #616[6168] - Change wait_xxx not to use locking
- Issue #615[6169] - Revert visibility 'fix' (fb0b6b8245dad1127b0c25ebafd9386b3945cca9)
- Issue #614[6170] - Fix Dataflow linker error
- Issue #613[6171] - find_here should throw an exception on failure
- Issue #612[6172] - Thread phase doesn't show up in debug mode
- Issue #611[6173] - Make stack guard pages configurable at runtime (initialization time)
- Issue #610[6174] - Co-Locate Components
- Issue #609[6175] - future_overhead
- Issue #608[6176] - `--hpx:list-counter-infos` problem
- Issue #607[6177] - Update Boost.Context based backend for coroutines
- Issue #606[6178] - 1d_wave_equation is not working

6156 https://github.com/STEllAR-GROUP/hpx/issues/637
6157 https://github.com/STEllAR-GROUP/hpx/issues/635
6158 https://github.com/STEllAR-GROUP/hpx/issues/633
6159 https://github.com/STEllAR-GROUP/hpx/issues/632
6160 https://github.com/STEllAR-GROUP/hpx/issues/631
6161 https://github.com/STEllAR-GROUP/hpx/issues/630
6162 https://github.com/STEllAR-GROUP/hpx/issues/629
6163 https://github.com/STEllAR-GROUP/hpx/issues/628
6164 https://github.com/STEllAR-GROUP/hpx/issues/625
6165 https://github.com/STEllAR-GROUP/hpx/issues/621
6166 https://github.com/STEllAR-GROUP/hpx/issues/618
6167 https://github.com/STEllAR-GROUP/hpx/issues/617
6168 https://github.com/STEllAR-GROUP/hpx/issues/616
6169 https://github.com/STEllAR-GROUP/hpx/issues/615
6170 https://github.com/STEllAR-GROUP/hpx/issues/614
6171 https://github.com/STEllAR-GROUP/hpx/issues/613
6172 https://github.com/STEllAR-GROUP/hpx/issues/612
6173 https://github.com/STEllAR-GROUP/hpx/issues/611
6174 https://github.com/STEllAR-GROUP/hpx/issues/610
6175 https://github.com/STEllAR-GROUP/hpx/issues/609
6176 https://github.com/STEllAR-GROUP/hpx/issues/608
6177 https://github.com/STEllAR-GROUP/hpx/issues/607
6178 https://github.com/STEllAR-GROUP/hpx/issues/606

- Issue #605[6179] - Any C++ function that has serializable arguments and a serializable return type should be remotable
- Issue #604[6180] - Connecting localities isn't working anymore
- Issue #603[6181] - Do not verify any ini entries read from a file
- Issue #602[6182] - Rename argument_size to type_size/ added implementation to get parcel size
- Issue #599[6183] - Enable locality specific command line options
- Issue #598[6184] - Need an API that accesses the performance counter reporting the system uptime
- Issue #597[6185] - compiling on ranger
- Issue #595[6186] - I need a place to store data in a thread self pointer
- Issue #594[6187] - 32/64 interoperability
- Issue #593[6188] - Warn if logging is disabled at compile time but requested at runtime
- Issue #592[6189] - Add optional argument value to `--hpx:list-counters` and `--hpx:list-counter-infos`
- Issue #591[6190] - Allow for wildcards in performance counter names specified with `--hpx:print-counter`
- Issue #590[6191] - Local promise semantic differences
- Issue #589[6192] - Create API to query performance counter names
- Issue #587[6193] - Add get_num_localities and get_num_threads to AGAS API
- Issue #586[6194] - Adjust local AGAS cache size based on number of localities
- Issue #585[6195] - Error while using counters in HPX
- Issue #584[6196] - counting argument size of actions, initial pass.
- Issue #581[6197] - Remove `RemoteResult` template parameter for `future`<>
- Issue #580[6198] - Add possibility to hook into actions
- Issue #578[6199] - Use angle brackets in HPX error dumps
- Issue #576[6200] - Exception incorrectly thrown when `--help` is used
- Issue #575[6201] - HPX(bad_component_type) with gcc 4.7.2 and boost 1.51

6179 https://github.com/STEllAR-GROUP/hpx/issues/605
6180 https://github.com/STEllAR-GROUP/hpx/issues/604
6181 https://github.com/STEllAR-GROUP/hpx/issues/603
6182 https://github.com/STEllAR-GROUP/hpx/issues/602
6183 https://github.com/STEllAR-GROUP/hpx/issues/599
6184 https://github.com/STEllAR-GROUP/hpx/issues/598
6185 https://github.com/STEllAR-GROUP/hpx/issues/597
6186 https://github.com/STEllAR-GROUP/hpx/issues/595
6187 https://github.com/STEllAR-GROUP/hpx/issues/594
6188 https://github.com/STEllAR-GROUP/hpx/issues/593
6189 https://github.com/STEllAR-GROUP/hpx/issues/592
6190 https://github.com/STEllAR-GROUP/hpx/issues/591
6191 https://github.com/STEllAR-GROUP/hpx/issues/590
6192 https://github.com/STEllAR-GROUP/hpx/issues/589
6193 https://github.com/STEllAR-GROUP/hpx/issues/587
6194 https://github.com/STEllAR-GROUP/hpx/issues/586
6195 https://github.com/STEllAR-GROUP/hpx/issues/585
6196 https://github.com/STEllAR-GROUP/hpx/issues/584
6197 https://github.com/STEllAR-GROUP/hpx/issues/581
6198 https://github.com/STEllAR-GROUP/hpx/issues/580
6199 https://github.com/STEllAR-GROUP/hpx/issues/578
6200 https://github.com/STEllAR-GROUP/hpx/issues/576
6201 https://github.com/STEllAR-GROUP/hpx/issues/575

- Issue #574[6202] - `--hpx:connect` command line parameter not working correctly
- Issue #571[6203] - `hpx::wait()` (callback version) should pass the future to the callback function
- Issue #570[6204] - `hpx::wait` should operate on `boost::arrays` and `std::lists`
- Issue #569[6205] - Add a logging sink for Android
- Issue #568[6206] - 2-argument version of `HPX_DEFINE_COMPONENT_ACTION`
- Issue #567[6207] - Connecting to a running HPX application works only once
- Issue #565[6208] - HPX doesn't shutdown properly
- Issue #564[6209] - Partial preprocessing of new component creation interface
- Issue #563[6210] - Add `hpx::start/hpx::stop` to avoid blocking main thread
- Issue #562[6211] - All command line arguments swallowed by hpx
- Issue #561[6212] - Boost.Tuple is not move aware
- Issue #558[6213] - `boost::shared_ptr`<> style semantics/syntax for client classes
- Issue #556[6214] - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
- Issue #555[6215] - `BOOST_FORCEINLINE` does not name a type
- Issue #554[6216] - Possible race condition in thread `get_id()`
- Issue #552[6217] - Move enable client_base
- Issue #550[6218] - Add stack size category 'huge'
- Issue #549[6219] - ShenEOS run seg-faults on single or distributed runs
- Issue #545[6220] - `AUTOGLOB` broken for add_hpx_component
- Issue #542[6221] - FindHPX_HDF5 still searches multiple times
- Issue #541[6222] - Quotes around application name in hpx::init
- Issue #539[6223] - Race conditition occurring with new lightweight threads
- Issue #535[6224] - hpx_run_tests.py exits with no error code when tests are missing

6202 https://github.com/STEllAR-GROUP/hpx/issues/574
6203 https://github.com/STEllAR-GROUP/hpx/issues/571
6204 https://github.com/STEllAR-GROUP/hpx/issues/570
6205 https://github.com/STEllAR-GROUP/hpx/issues/569
6206 https://github.com/STEllAR-GROUP/hpx/issues/568
6207 https://github.com/STEllAR-GROUP/hpx/issues/567
6208 https://github.com/STEllAR-GROUP/hpx/issues/565
6209 https://github.com/STEllAR-GROUP/hpx/issues/564
6210 https://github.com/STEllAR-GROUP/hpx/issues/563
6211 https://github.com/STEllAR-GROUP/hpx/issues/562
6212 https://github.com/STEllAR-GROUP/hpx/issues/561
6213 https://github.com/STEllAR-GROUP/hpx/issues/558
6214 https://github.com/STEllAR-GROUP/hpx/issues/556
6215 https://github.com/STEllAR-GROUP/hpx/issues/555
6216 https://github.com/STEllAR-GROUP/hpx/issues/554
6217 https://github.com/STEllAR-GROUP/hpx/issues/552
6218 https://github.com/STEllAR-GROUP/hpx/issues/550
6219 https://github.com/STEllAR-GROUP/hpx/issues/549
6220 https://github.com/STEllAR-GROUP/hpx/issues/545
6221 https://github.com/STEllAR-GROUP/hpx/issues/542
6222 https://github.com/STEllAR-GROUP/hpx/issues/541
6223 https://github.com/STEllAR-GROUP/hpx/issues/539
6224 https://github.com/STEllAR-GROUP/hpx/issues/535

- Issue #530[6225] - Thread description(<unknown>) in logs
- Issue #523[6226] - Make thread objects more lightweight
- Issue #521[6227] - `hpx::error_code` is not usable for lightweight error handling
- Issue #520[6228] - Add full user environment to HPX logs
- Issue #519[6229] - Build succeeds, running fails
- Issue #517[6230] - Add a guard page to linux coroutine stacks
- Issue #516[6231] - hpx::thread::detach suspends while holding locks, leads to hang in debug
- Issue #514[6232] - Preprocessed headers for <hpx/apply.hpp> don't compile
- Issue #513[6233] - Buildbot configuration problem
- Issue #512[6234] - Implement action based stack size customization
- Issue #511[6235] - Move action priority into a separate type trait
- Issue #510[6236] - trunk broken
- Issue #507[6237] - no matching function for call to `boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(hpx::thre`
- Issue #505[6238] - undefined_symbol regression test currently failing
- Issue #502[6239] - Adding OpenCL and OCLM support to HPX for Windows and Linux
- Issue #501[6240] - find_package(HPX) sets cmake output variables
- Issue #500[6241] - wait_any/wait_all are badly named
- Issue #499[6242] - Add support for disabling pbs support in pbs runs
- Issue #498[6243] - Error during no-cache runs
- Issue #496[6244] - Add partial preprocessing support to cmake
- Issue #495[6245] - Support HPX modules exporting startup/shutdown functions only
- Issue #494[6246] - Allow modules to specify when to run startup/shutdown functions
- Issue #493[6247] - Avoid constructing a string in make_success_code

6225 https://github.com/STEllAR-GROUP/hpx/issues/530
6226 https://github.com/STEllAR-GROUP/hpx/issues/523
6227 https://github.com/STEllAR-GROUP/hpx/issues/521
6228 https://github.com/STEllAR-GROUP/hpx/issues/520
6229 https://github.com/STEllAR-GROUP/hpx/issues/519
6230 https://github.com/STEllAR-GROUP/hpx/issues/517
6231 https://github.com/STEllAR-GROUP/hpx/issues/516
6232 https://github.com/STEllAR-GROUP/hpx/issues/514
6233 https://github.com/STEllAR-GROUP/hpx/issues/513
6234 https://github.com/STEllAR-GROUP/hpx/issues/512
6235 https://github.com/STEllAR-GROUP/hpx/issues/511
6236 https://github.com/STEllAR-GROUP/hpx/issues/510
6237 https://github.com/STEllAR-GROUP/hpx/issues/507
6238 https://github.com/STEllAR-GROUP/hpx/issues/505
6239 https://github.com/STEllAR-GROUP/hpx/issues/502
6240 https://github.com/STEllAR-GROUP/hpx/issues/501
6241 https://github.com/STEllAR-GROUP/hpx/issues/500
6242 https://github.com/STEllAR-GROUP/hpx/issues/499
6243 https://github.com/STEllAR-GROUP/hpx/issues/498
6244 https://github.com/STEllAR-GROUP/hpx/issues/496
6245 https://github.com/STEllAR-GROUP/hpx/issues/495
6246 https://github.com/STEllAR-GROUP/hpx/issues/494
6247 https://github.com/STEllAR-GROUP/hpx/issues/493

- Issue #492[6248] - Performance counter creation is no longer synchronized at startup
- Issue #491[6249] - Performance counter creation is no longer synchronized at startup
- Issue #490[6250] - Sheneos on_completed_bulk seg fault in distributed
- Issue #489[6251] - compiling issue with g++44
- Issue #488[6252] - Adding OpenCL and OCLM support to HPX for the MSVC platform
- Issue #487[6253] - FindHPX.cmake problems
- Issue #485[6254] - Change distributing_factory and binpacking_factory to use bulk creation
- Issue #484[6255] - Change `HPX_DONT_USE_PREPROCESSED_FILES` to `HPX_USE_PREPROCESSED_FILES`
- Issue #483[6256] - Memory counter for Windows
- Issue #479[6257] - strange errors appear when requesting performance counters on multiple nodes
- Issue #477[6258] - Create (global) timer for multi-threaded measurements
- Issue #472[6259] - Add partial preprocessing using Wave
- Issue #471[6260] - Segfault stack traces don't show up in release
- Issue #468[6261] - External projects need to link with internal components
- Issue #462[6262] - Startup/shutdown functions are called more than once
- Issue #458[6263] - Consolidate hpx::util::high_resolution_timer and `hpx::util::high_resolution_clock`
- Issue #457[6264] - index out of bounds in `allgather_and_gate` on 4 cores or more
- Issue #448[6265] - Make HPX compile with clang
- Issue #447[6266] - 'make tests' should execute tests on local installation
- Issue #446[6267] - Remove SVN-related code from the codebase
- Issue #444[6268] - race condition in smp
- Issue #441[6269] - Patched Boost.Serialization headers should only be installed if needed
- Issue #439[6270] - Components using `HPX_REGISTER_STARTUP_MODULE` fail to compile with MSVC

[6248] https://github.com/STEllAR-GROUP/hpx/issues/492
[6249] https://github.com/STEllAR-GROUP/hpx/issues/491
[6250] https://github.com/STEllAR-GROUP/hpx/issues/490
[6251] https://github.com/STEllAR-GROUP/hpx/issues/489
[6252] https://github.com/STEllAR-GROUP/hpx/issues/488
[6253] https://github.com/STEllAR-GROUP/hpx/issues/487
[6254] https://github.com/STEllAR-GROUP/hpx/issues/485
[6255] https://github.com/STEllAR-GROUP/hpx/issues/484
[6256] https://github.com/STEllAR-GROUP/hpx/issues/483
[6257] https://github.com/STEllAR-GROUP/hpx/issues/479
[6258] https://github.com/STEllAR-GROUP/hpx/issues/477
[6259] https://github.com/STEllAR-GROUP/hpx/issues/472
[6260] https://github.com/STEllAR-GROUP/hpx/issues/471
[6261] https://github.com/STEllAR-GROUP/hpx/issues/468
[6262] https://github.com/STEllAR-GROUP/hpx/issues/462
[6263] https://github.com/STEllAR-GROUP/hpx/issues/458
[6264] https://github.com/STEllAR-GROUP/hpx/issues/457
[6265] https://github.com/STEllAR-GROUP/hpx/issues/448
[6266] https://github.com/STEllAR-GROUP/hpx/issues/447
[6267] https://github.com/STEllAR-GROUP/hpx/issues/446
[6268] https://github.com/STEllAR-GROUP/hpx/issues/444
[6269] https://github.com/STEllAR-GROUP/hpx/issues/441
[6270] https://github.com/STEllAR-GROUP/hpx/issues/439

- Issue #436[6271] - Verify that no locks are being held while threads are suspended
- Issue #435[6272] - Installing HPX should not clobber existing Boost installation
- Issue #434[6273] - Logging external component failed (Boost 1.50)
- Issue #433[6274] - Runtime crash when building all examples
- Issue #432[6275] - Dataflow hangs on 512 cores/64 nodes
- Issue #430[6276] - Problem with distributing factory
- Issue #424[6277] - File paths referring to XSL-files need to be properly escaped
- Issue #417[6278] - Make dataflow LCOs work out of the box by using partial preprocessing
- Issue #413[6279] - hpx_svnversion.py fails on Windows
- Issue #412[6280] - Make hpx::error_code equivalent to hpx::exception
- Issue #398[6281] - HPX clobbers out-of-tree application specific CMake variables (specifically `CMAKE_BUILD_TYPE`)
- Issue #394[6282] - Remove code generating random port numbers for network
- Issue #378[6283] - ShenEOS scaling issues
- Issue #354[6284] - Create a coroutines wrapper for Boost.Context
- Issue #349[6285] - Commandline option `--localities=N/-lN` should be necessary only on AGAS locality
- Issue #334[6286] - Add auto_index support to cmake based documentation toolchain
- Issue #318[6287] - Network benchmarks
- Issue #317[6288] - Implement network performance counters
- Issue #310[6289] - Duplicate logging entries
- Issue #230[6290] - Add compile time option to disable thread debugging info
- Issue #171[6291] - Add an INI option to turn off deadlock detection independently of logging
- Issue #170[6292] - OSHL internal counters are incorrect
- Issue #103[6293] - Better diagnostics for multiple component/action registerations under the same name

6271 https://github.com/STEllAR-GROUP/hpx/issues/436
6272 https://github.com/STEllAR-GROUP/hpx/issues/435
6273 https://github.com/STEllAR-GROUP/hpx/issues/434
6274 https://github.com/STEllAR-GROUP/hpx/issues/433
6275 https://github.com/STEllAR-GROUP/hpx/issues/432
6276 https://github.com/STEllAR-GROUP/hpx/issues/430
6277 https://github.com/STEllAR-GROUP/hpx/issues/424
6278 https://github.com/STEllAR-GROUP/hpx/issues/417
6279 https://github.com/STEllAR-GROUP/hpx/issues/413
6280 https://github.com/STEllAR-GROUP/hpx/issues/412
6281 https://github.com/STEllAR-GROUP/hpx/issues/398
6282 https://github.com/STEllAR-GROUP/hpx/issues/394
6283 https://github.com/STEllAR-GROUP/hpx/issues/378
6284 https://github.com/STEllAR-GROUP/hpx/issues/354
6285 https://github.com/STEllAR-GROUP/hpx/issues/349
6286 https://github.com/STEllAR-GROUP/hpx/issues/334
6287 https://github.com/STEllAR-GROUP/hpx/issues/318
6288 https://github.com/STEllAR-GROUP/hpx/issues/317
6289 https://github.com/STEllAR-GROUP/hpx/issues/310
6290 https://github.com/STEllAR-GROUP/hpx/issues/230
6291 https://github.com/STEllAR-GROUP/hpx/issues/171
6292 https://github.com/STEllAR-GROUP/hpx/issues/170
6293 https://github.com/STEllAR-GROUP/hpx/issues/103

- Issue #48[6294] - Support for Darwin (Xcode + Clang)
- Issue #21[6295] - Build fails with GCC 4.6

HPX V0.9.0 (Jul 5, 2012)

We have had roughly 800 commits since the last release and we have closed approximately 80 tickets (bugs, feature requests, etc.).

General changes

- Significant improvements made to the usability of *HPX* in large-scale, distributed environments.
- Renamed `hpx::lcos::packaged_task` to `hpx::lcos::packaged_action` to reflect the semantic differences to a packaged_task as defined by the C++11 Standard[6296].
- *HPX* now exposes `hpx::thread` which is compliant to the C++11 std::thread type except that it (purely locally) represents an *HPX* thread. This new type does not expose any of the remote capabilities of the underlying *HPX*-thread implementation.
- The type `hpx::lcos::future` is now compliant to the C++11 std::future<> type. This type can be used to synchronize both, local and remote operations. In both cases the control flow will 'return' to the future in order to trigger any continuation.
- The types `hpx::lcos::local::promise` and `hpx::lcos::local::packaged_task` are now compliant to the C++11 `std::promise<>` and `std::packaged_task<>` types. These can be used to create a future representing local work only. Use the types `hpx::lcos::promise` and `hpx::lcos::packaged_action` to wrap any (possibly remote) action into a future.
- `hpx::thread` and `hpx::lcos::future` are now cancelable.
- Added support for sequential and logic composition of `hpx::lcos::futures`. The member function `hpx::lcos::future::when` permits futures to be sequentially composed. The helper functions `hpx::wait_all`, `hpx::wait_any`, and `hpx::wait_n` can be used to wait for more than one future at a time.
- *HPX* now exposes `hpx::apply` and `hpx::async` as the preferred way of creating (or invoking) any deferred work. These functions are usable with various types of functions, function objects, and actions and provide a uniform way to spawn deferred tasks.
- *HPX* now utilizes `hpx::util::bind` to (partially) bind local functions and function objects, and also actions. Remote bound actions can have placeholders as well.
- *HPX* continuations are now fully polymorphic. The class `hpx::actions::forwarding_continuation` is an example of how the user can write is own types of continuations. It can be used to execute any function as an continuation of a particular action.
- Reworked the action invocation API to be fully conformant to normal functions. Actions can now be invoked using `hpx::apply`, `hpx::async`, or using the `operator()` implemented on actions. Actions themselves can now be cheaply instantiated as they do not have any members anymore.
- Reworked the lazy action invocation API. Actions can now be directly bound using `hpx::util::bind` by passing an action instance as the first argument.
- A minimal *HPX* program now looks like this:

[6294] https://github.com/STEllAR-GROUP/hpx/issues/48
[6295] https://github.com/STEllAR-GROUP/hpx/issues/21
[6296] http://www.open-std.org/jtc1/sc22/wg21

```
#include <hpx/hpx_init.hpp>

int hpx_main()
{
    return hpx::finalize();
}

int main()
{
    return hpx::init();
}
```

This removes the immediate dependency on the Boost.Program_options[6297] library.

Note: This minimal version of an *HPX* program does not support any of the default command line arguments (such as –help, or command line options related to PBS). It is suggested to always pass `argc` and `argv` to *HPX* as shown in the example below.

- In order to support those, but still not to depend on Boost.Program_options[6298], the minimal program can be written as:

```
#include <hpx/hpx_init.hpp>

// The arguments for hpx_main can be left off, which very similar to the
// behavior of ``main()`` as defined by C++.
int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);
}
```

- Added performance counters exposing the number of component instances which are alive on a given locality.
- Added performance counters exposing then number of messages sent and received, the number of parcels sent and received, the number of bytes sent and received, the overall time required to send and receive data, and the overall time required to serialize and deserialize the data.
- Added a new component: `hpx::components::binpacking_factory` which is equivalent to the existing `hpx::components::distributing_factory` component, except that it equalizes the overall population of the components to create. It exposes two factory methods, one based on the number of existing instances of the component type to create, and one based on an arbitrary performance counter which will be queried for all relevant localities.
- Added API functions allowing to access elements of the diagnostic information embedded in the given exception: *`hpx::get_locality_id`*, `hpx::get_host_name`, `hpx::get_process_id`, `hpx::get_function_name`, `hpx::get_file_name`, `hpx::get_line_number`, `hpx::get_os_thread`, `hpx::get_thread_id`, and `hpx::get_thread_description`.

6297 https://www.boost.org/doc/html/program_options.html
6298 https://www.boost.org/doc/html/program_options.html

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #71[6299] - GIDs that are not serialized via `handle_gid<>` should raise an exception
- Issue #105[6300] - Allow for `hpx::util::function`s to be registered in the AGAS symbolic namespace
- Issue #107[6301] - Nasty threadmanger race condition (reproducible in sheneos_test)
- Issue #108[6302] - Add millisecond resolution to *HPX* logs on Linux
- Issue #110[6303] - Shutdown hang in distributed with release build
- Issue #116[6304] - Don't use TSS for the applier and runtime pointers
- Issue #162[6305] - Move local synchronous execution shortcut from hpx::function to the applier
- Issue #172[6306] - Cache sources in CMake and check if they change manually
- Issue #178[6307] - Add an INI option to turn off ranged-based AGAS caching
- Issue #187[6308] - Support for disabling performance counter deployment
- Issue #202[6309] - Support for sending performance counter data to a specific file
- Issue #218[6310] - boost.coroutines allows different stack sizes, but stack pool is unaware of this
- Issue #231[6311] - Implement movable `boost::bind`
- Issue #232[6312] - Implement movable `boost::function`
- Issue #236[6313] - Allow binding `hpx::util::function` to actions
- Issue #239[6314] - Replace `hpx::function` with `hpx::util::function`
- Issue #240[6315] - Can't specify RemoteResult with lcos::async
- Issue #242[6316] - REGISTER_TEMPLATE support for plain actions
- Issue #243[6317] - `handle_gid<>` support for `hpx::util::function`
- Issue #245[6318] - `*_c_cache code` throws an exception if the queried GID is not in the local cache
- Issue #246[6319] - Undefined references in dataflow/adaptive1d example

6299 https://github.com/STEllAR-GROUP/hpx/issues/71
6300 https://github.com/STEllAR-GROUP/hpx/issues/105
6301 https://github.com/STEllAR-GROUP/hpx/issues/107
6302 https://github.com/STEllAR-GROUP/hpx/issues/108
6303 https://github.com/STEllAR-GROUP/hpx/issues/110
6304 https://github.com/STEllAR-GROUP/hpx/issues/116
6305 https://github.com/STEllAR-GROUP/hpx/issues/162
6306 https://github.com/STEllAR-GROUP/hpx/issues/172
6307 https://github.com/STEllAR-GROUP/hpx/issues/178
6308 https://github.com/STEllAR-GROUP/hpx/issues/187
6309 https://github.com/STEllAR-GROUP/hpx/issues/202
6310 https://github.com/STEllAR-GROUP/hpx/issues/218
6311 https://github.com/STEllAR-GROUP/hpx/issues/231
6312 https://github.com/STEllAR-GROUP/hpx/issues/232
6313 https://github.com/STEllAR-GROUP/hpx/issues/236
6314 https://github.com/STEllAR-GROUP/hpx/issues/239
6315 https://github.com/STEllAR-GROUP/hpx/issues/240
6316 https://github.com/STEllAR-GROUP/hpx/issues/242
6317 https://github.com/STEllAR-GROUP/hpx/issues/243
6318 https://github.com/STEllAR-GROUP/hpx/issues/245
6319 https://github.com/STEllAR-GROUP/hpx/issues/246

- Issue #252[6320] - Problems configuring sheneos with CMake
- Issue #254[6321] - Lifetime of components doesn't end when client goes out of scope
- Issue #259[6322] - CMake does not detect that MSVC10 has lambdas
- Issue #260[6323] - io_service_pool segfault
- Issue #261[6324] - Late parcel executed outside of pxthread
- Issue #263[6325] - Cannot select allocator with CMake
- Issue #264[6326] - Fix allocator select
- Issue #267[6327] - Runtime error for hello_world
- Issue #269[6328] - pthread_affinity_np test fails to compile
- Issue #270[6329] - Compiler noise due to -Wcast-qual
- Issue #275[6330] - Problem with configuration tests/include paths on Gentoo
- Issue #325[6331] - Sheneos is 200-400 times slower than the fortran equivalent
- Issue #331[6332] - `hpx::init` and `hpx_main()` should not depend on program_options
- Issue #333[6333] - Add doxygen support to CMake for doc toolchain
- Issue #340[6334] - Performance counters for parcels
- Issue #346[6335] - Component loading error when running hello_world in distributed on MSVC2010
- Issue #362[6336] - Missing initializer error
- Issue #363[6337] - Parcel port serialization error
- Issue #366[6338] - Parcel buffering leads to types incompatible exception
- Issue #368[6339] - Scalable alternative to rand() needed for *HPX*
- Issue #369[6340] - IB over IP is substantially slower than just using standard TCP/IP
- Issue #374[6341] - `hpx::lcos::wait` should work with dataflows and arbitrary classes meeting the future interface
- Issue #375[6342] - Conflicting/ambiguous overloads of `hpx::lcos::wait`

6320 https://github.com/STEllAR-GROUP/hpx/issues/252
6321 https://github.com/STEllAR-GROUP/hpx/issues/254
6322 https://github.com/STEllAR-GROUP/hpx/issues/259
6323 https://github.com/STEllAR-GROUP/hpx/issues/260
6324 https://github.com/STEllAR-GROUP/hpx/issues/261
6325 https://github.com/STEllAR-GROUP/hpx/issues/263
6326 https://github.com/STEllAR-GROUP/hpx/issues/264
6327 https://github.com/STEllAR-GROUP/hpx/issues/267
6328 https://github.com/STEllAR-GROUP/hpx/issues/269
6329 https://github.com/STEllAR-GROUP/hpx/issues/270
6330 https://github.com/STEllAR-GROUP/hpx/issues/275
6331 https://github.com/STEllAR-GROUP/hpx/issues/325
6332 https://github.com/STEllAR-GROUP/hpx/issues/331
6333 https://github.com/STEllAR-GROUP/hpx/issues/333
6334 https://github.com/STEllAR-GROUP/hpx/issues/340
6335 https://github.com/STEllAR-GROUP/hpx/issues/346
6336 https://github.com/STEllAR-GROUP/hpx/issues/362
6337 https://github.com/STEllAR-GROUP/hpx/issues/363
6338 https://github.com/STEllAR-GROUP/hpx/issues/366
6339 https://github.com/STEllAR-GROUP/hpx/issues/368
6340 https://github.com/STEllAR-GROUP/hpx/issues/369
6341 https://github.com/STEllAR-GROUP/hpx/issues/374
6342 https://github.com/STEllAR-GROUP/hpx/issues/375

- Issue #376[6343] - Find_HPX.cmake should set CMake variable HPX_FOUND for out of tree builds
- Issue #377[6344] - ShenEOS interpolate bulk and interpolate_one_bulk are broken
- Issue #379[6345] - Add support for distributed runs under SLURM
- Issue #382[6346] - _Unwind_Word not declared in boost.backtrace
- Issue #387[6347] - Doxygen should look only at list of specified files
- Issue #388[6348] - Running `make install` on an out-of-tree application is broken
- Issue #391[6349] - Out-of-tree application segfaults when running in qsub
- Issue #392[6350] - Remove HPX_NO_INSTALL option from cmake build system
- Issue #396[6351] - Pragma related warnings when compiling with older gcc versions
- Issue #399[6352] - Out of tree component build problems
- Issue #400[6353] - Out of source builds on Windows: linker should not receive compiler flags
- Issue #401[6354] - Out of source builds on Windows: components need to be linked with hpx_serialization
- Issue #404[6355] - gfortran fails to link automatically when fortran files are present
- Issue #405[6356] - Inability to specify linking order for external libraries
- Issue #406[6357] - Adapt action limits such that dataflow applications work without additional defines
- Issue #415[6358] - `locality_results` is not a member of `hpx::components::server`
- Issue #425[6359] - Breaking changes to `traits::*result` wrt `std::vector<id_type>`
- Issue #426[6360] - AUTOGLOB needs to be updated to support fortran

HPX V0.8.1 (Apr 21, 2012)

This is a point release including important bug fixes for *HPX V0.8.0 (Mar 23, 2012)*.

6343 https://github.com/STEllAR-GROUP/hpx/issues/376
6344 https://github.com/STEllAR-GROUP/hpx/issues/377
6345 https://github.com/STEllAR-GROUP/hpx/issues/379
6346 https://github.com/STEllAR-GROUP/hpx/issues/382
6347 https://github.com/STEllAR-GROUP/hpx/issues/387
6348 https://github.com/STEllAR-GROUP/hpx/issues/388
6349 https://github.com/STEllAR-GROUP/hpx/issues/391
6350 https://github.com/STEllAR-GROUP/hpx/issues/392
6351 https://github.com/STEllAR-GROUP/hpx/issues/396
6352 https://github.com/STEllAR-GROUP/hpx/issues/399
6353 https://github.com/STEllAR-GROUP/hpx/issues/400
6354 https://github.com/STEllAR-GROUP/hpx/issues/401
6355 https://github.com/STEllAR-GROUP/hpx/issues/404
6356 https://github.com/STEllAR-GROUP/hpx/issues/405
6357 https://github.com/STEllAR-GROUP/hpx/issues/406
6358 https://github.com/STEllAR-GROUP/hpx/issues/415
6359 https://github.com/STEllAR-GROUP/hpx/issues/425
6360 https://github.com/STEllAR-GROUP/hpx/issues/426

General changes

- *HPX* does not need to be installed anymore to be functional.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this point release:

- Issue #295[6361] - Don't require install path to be known at compile time.
- Issue #371[6362] - Add hpx iostreams to standard build.
- Issue #384[6363] - Fix compilation with GCC 4.7.
- Issue #390[6364] - Remove keep_factory_alive startup call from ShenEOS; add shutdown call to H5close.
- Issue #393[6365] - Thread affinity control is broken.

Bug fixes (commits)

Here is a list of the important commits included in this point release:

- r7642 - External: Fix backtrace memory violation.
- **r7775 - Components: Fix symbol visibility bug with component startup** providers. This prevents one components providers from overriding another components.
- r7778 - Components: Fix startup/shutdown provider shadowing issues.

HPX V0.8.0 (Mar 23, 2012)

We have had roughly 1000 commits since the last release and we have closed approximately 70 tickets (bugs, feature requests, etc.).

General changes

- Improved PBS support, allowing for arbitrary naming schemes of node-hostnames.
- Finished verification of the reference counting framework.
- Implemented decrement merging logic to optimize the distributed reference counting system.
- Restructured the LCO framework. Renamed `hpx::lcos::eager_future<>` and `hpx::lcos::lazy_future<>` into `hpx::lcos::packaged_task` and `hpx::lcos::deferred_packaged_task`. Split `hpx::lcos::promise` into `hpx::lcos::packaged_task` and `hpx::lcos::future`. Added 'local' futures (in namespace `hpx::lcos::local`).
- Improved the general performance of local and remote action invocations. This (under certain circumstances) drastically reduces the number of copies created for each of the parameters and return values.

[6361] https://github.com/STEllAR-GROUP/hpx/issues/295
[6362] https://github.com/STEllAR-GROUP/hpx/issues/371
[6363] https://github.com/STEllAR-GROUP/hpx/issues/384
[6364] https://github.com/STEllAR-GROUP/hpx/issues/390
[6365] https://github.com/STEllAR-GROUP/hpx/issues/393

- Reworked the performance counter framework. Performance counters are now created only when needed, which reduces the overall resource requirements. The new framework allows for much more flexible creation and management of performance counters. The new sine example application demonstrates some of the capabilities of the new infrastructure.
- Added a buildbot-based continuous build system which gives instant, automated feedback on each commit to SVN.
- Added more automated tests to verify proper functioning of *HPX*.
- Started to create documentation for *HPX* and its API.
- Added documentation toolchain to the build system.
- Added dataflow LCO.
- Changed default *HPX* command line options to have `hpx:` prefix. For instance, the former option `--threads` is now `--hpx:threads`. This has been done to make ambiguities with possible application specific command line options as unlikely as possible. See the section *HPX Command Line Options* for a full list of available options.
- Added the possibility to define command line aliases. The former short (one-letter) command line options have been predefined as aliases for backwards compatibility. See the section *HPX Command Line Options* for a detailed description of command line option aliasing.
- Network connections are now cached based on the connected host. The number of simultaneous connections to a particular host is now limited. Parcels are buffered and bundled if all connections are in use.
- Added more refined thread affinity control. This is based on the external library Portable Hardware Locality (HWLOC).
- Improved support for Windows builds with CMake.
- Added support for components to register their own command line options.
- Added the possibility to register custom startup/shutdown functions for any component. These functions are guaranteed to be executed by an *HPX* thread.
- Added two new experimental thread schedulers: hierarchy_scheduler and periodic_priority_scheduler. These can be activated by using the command line options `--hpx:queuing=hierarchy` or `--hpx:queuing=periodic`.

Example applications

- Graph500 performance benchmark[6366] (thanks to Matthew Anderson for contributing this application).
- GTC (Gyrokinetic Toroidal Code)[6367]: a skeleton for particle in cell type codes.
- Random Memory Access: an example demonstrating random memory accesses in a large array
- ShenEOS example[6368], demonstrating partitioning of large read-only data structures and exposing an interpolation API.
- Sine performance counter demo.
- Accumulator examples demonstrating how to write and use *HPX* components.
- Quickstart examples (like hello_world, fibonacci, quicksort, factorial, etc.) demonstrating simple *HPX* concepts which introduce some of the concepts in *HPX*.
- Load balancing and work stealing demos.

[6366] http://www.graph500.org/
[6367] http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/gtc/
[6368] http://stellarcollapse.org/equationofstate

API changes

- Moved all local LCOs into a separate namespace `hpx::lcos::local` (for instance, `hpx::lcos::local_mutex` is now `hpx::lcos::local::mutex`).
- Replaced `hpx::actions::function` with `hpx::util::function`. Cleaned up related code.
- Removed `hpx::traits::handle_gid` and moved handling of global reference counts into the corresponding serialization code.
- Changed terminology: `prefix` is now called `locality_id`, renamed the corresponding API functions (such as `hpx::get_prefix`, which is now called `hpx::get_locality_id`).
- Adding *hpx::find_remote_localities*, and *hpx::get_num_localities*.
- Changed performance counter naming scheme to make it more bash friendly. The new performance counter naming scheme is now

```
/object{parentname#parentindex/instance#index}/counter#parameters
```

- Added `hpx::get_worker_thread_num` replacing `hpx::threadmanager_base::get_thread_num`.
- Renamed `hpx::get_num_os_threads` to `hpx::get_os_threads_count`.
- Added `hpx::threads::get_thread_count`.
- Restructured the Futures sub-system, renaming types in accordance with the terminology used by the C++11 ISO standard.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #31[6369] - Specialize handle_gid<> for examples and tests
- Issue #72[6370] - Fix AGAS reference counting
- Issue #104[6371] - heartbeat throws an exception when decrefing the performance counter it's watching
- Issue #111[6372] - throttle causes an exception on the target application
- Issue #142[6373] - One failed component loading causes an unrelated component to fail
- Issue #165[6374] - Remote exception propagation bug in AGAS reference counting test
- Issue #186[6375] - Test credit exhaustion/splitting (e.g. prepare_gid and symbol NS)
- Issue #188[6376] - Implement remaining AGAS reference counting test cases
- Issue #258[6377] - No type checking of GIDs in stubs classes
- Issue #271[6378] - Seg fault/shared pointer assertion in distributed code

[6369] https://github.com/STEllAR-GROUP/hpx/issues/31
[6370] https://github.com/STEllAR-GROUP/hpx/issues/72
[6371] https://github.com/STEllAR-GROUP/hpx/issues/104
[6372] https://github.com/STEllAR-GROUP/hpx/issues/111
[6373] https://github.com/STEllAR-GROUP/hpx/issues/142
[6374] https://github.com/STEllAR-GROUP/hpx/issues/165
[6375] https://github.com/STEllAR-GROUP/hpx/issues/186
[6376] https://github.com/STEllAR-GROUP/hpx/issues/188
[6377] https://github.com/STEllAR-GROUP/hpx/issues/258
[6378] https://github.com/STEllAR-GROUP/hpx/issues/271

- Issue #281[6379] - CMake options need descriptive text
- Issue #283[6380] - AGAS caching broken (gva_cache needs to be rewritten with ICL)
- Issue #285[6381] - HPX_INSTALL root directory not the same as CMAKE_INSTALL_PREFIX
- Issue #286[6382] - New segfault in dataflow applications
- Issue #289[6383] - Exceptions should only be logged if not handled
- Issue #290[6384] - c++11 tests failure
- Issue #293[6385] - Build target for component libraries
- Issue #296[6386] - Compilation error with Boost V1.49rc1
- Issue #298[6387] - Illegal instructions on termination
- Issue #299[6388] - gravity aborts with multiple threads
- Issue #301[6389] - Build error with Boost trunk
- Issue #303[6390] - Logging assertion failure in distributed runs
- Issue #304[6391] - Exception 'what' strings are lost when exceptions from decode_parcel are reported
- Issue #306[6392] - Performance counter user interface issues
- Issue #307[6393] - Logging exception in distributed runs
- Issue #308[6394] - Logging deadlocks in distributed
- Issue #309[6395] - Reference counting test failures and exceptions
- Issue #311[6396] - Merge AGAS remote_interface with the runtime_support object
- Issue #314[6397] - Object tracking for id_types
- Issue #315[6398] - Remove handle_gid and handle credit splitting in id_type serialization
- Issue #320[6399] - applier::get_locality_id() should return an error value (or throw an exception)
- Issue #321[6400] - Optimization for id_types which are never split should be restored
- Issue #322[6401] - Command line processing ignored with Boost 1.47.0

[6379] https://github.com/STEllAR-GROUP/hpx/issues/281
[6380] https://github.com/STEllAR-GROUP/hpx/issues/283
[6381] https://github.com/STEllAR-GROUP/hpx/issues/285
[6382] https://github.com/STEllAR-GROUP/hpx/issues/286
[6383] https://github.com/STEllAR-GROUP/hpx/issues/289
[6384] https://github.com/STEllAR-GROUP/hpx/issues/290
[6385] https://github.com/STEllAR-GROUP/hpx/issues/293
[6386] https://github.com/STEllAR-GROUP/hpx/issues/296
[6387] https://github.com/STEllAR-GROUP/hpx/issues/298
[6388] https://github.com/STEllAR-GROUP/hpx/issues/299
[6389] https://github.com/STEllAR-GROUP/hpx/issues/301
[6390] https://github.com/STEllAR-GROUP/hpx/issues/303
[6391] https://github.com/STEllAR-GROUP/hpx/issues/304
[6392] https://github.com/STEllAR-GROUP/hpx/issues/306
[6393] https://github.com/STEllAR-GROUP/hpx/issues/307
[6394] https://github.com/STEllAR-GROUP/hpx/issues/308
[6395] https://github.com/STEllAR-GROUP/hpx/issues/309
[6396] https://github.com/STEllAR-GROUP/hpx/issues/311
[6397] https://github.com/STEllAR-GROUP/hpx/issues/314
[6398] https://github.com/STEllAR-GROUP/hpx/issues/315
[6399] https://github.com/STEllAR-GROUP/hpx/issues/320
[6400] https://github.com/STEllAR-GROUP/hpx/issues/321
[6401] https://github.com/STEllAR-GROUP/hpx/issues/322

- Issue #323[6402] - Credit exhaustion causes object to stay alive
- Issue #324[6403] - Duplicate exception messages
- Issue #326[6404] - Integrate Quickbook with CMake
- Issue #329[6405] - –help and –version should still work
- Issue #330[6406] - Create pkg-config files
- Issue #337[6407] - Improve usability of performance counter timestamps
- Issue #338[6408] - Non-std exceptions deriving from std::exceptions in tfunc may be sliced
- Issue #339[6409] - Decrease the number of send_pending_parcels threads
- Issue #343[6410] - Dynamically setting the stack size doesn't work
- Issue #351[6411] - 'make install' does not update documents
- Issue #353[6412] - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs
- Issue #355[6413] - 'make' doesn't do anything after correct configuration
- Issue #356[6414] - Don't use `hpx::util::static_` in topology code
- Issue #359[6415] - Infinite recursion in hpx::tuple serialization
- Issue #361[6416] - Add compile time option to disable logging completely
- Issue #364[6417] - Installation seriously broken in r7443

HPX V0.7.0 (Dec 12, 2011)

We have had roughly 1000 commits since the last release and we have closed approximately 120 tickets (bugs, feature requests, etc.).

6402 https://github.com/STEllAR-GROUP/hpx/issues/323
6403 https://github.com/STEllAR-GROUP/hpx/issues/324
6404 https://github.com/STEllAR-GROUP/hpx/issues/326
6405 https://github.com/STEllAR-GROUP/hpx/issues/329
6406 https://github.com/STEllAR-GROUP/hpx/issues/330
6407 https://github.com/STEllAR-GROUP/hpx/issues/337
6408 https://github.com/STEllAR-GROUP/hpx/issues/338
6409 https://github.com/STEllAR-GROUP/hpx/issues/339
6410 https://github.com/STEllAR-GROUP/hpx/issues/343
6411 https://github.com/STEllAR-GROUP/hpx/issues/351
6412 https://github.com/STEllAR-GROUP/hpx/issues/353
6413 https://github.com/STEllAR-GROUP/hpx/issues/355
6414 https://github.com/STEllAR-GROUP/hpx/issues/356
6415 https://github.com/STEllAR-GROUP/hpx/issues/359
6416 https://github.com/STEllAR-GROUP/hpx/issues/361
6417 https://github.com/STEllAR-GROUP/hpx/issues/364

General changes

- Completely removed code related to deprecated AGAS V1, started to work on AGAS V2.1.
- Started to clean up and streamline the exposed APIs (see 'API changes' below for more details).
- Revamped and unified performance counter framework, added a lot of new performance counter instances for monitoring of a diverse set of internal *HPX* parameters (queue lengths, access statistics, etc.).
- Improved general error handling and logging support.
- Fixed several race conditions, improved overall stability, decreased memory footprint, improved overall performance (major optimizations include native TLS support and ranged-based AGAS caching).
- Added support for running *HPX* applications with PBS.
- Many updates to the build system, added support for gcc 4.5.x and 4.6.x, added C++11 support.
- Many updates to default command line options.
- Added many tests, set up buildbot for continuous integration testing.
- Better shutdown handling of distributed applications.

Example applications

- quickstart/factorial and quickstart/fibonacci, future-recursive parallel algorithms.
- quickstart/hello_world, distributed hello world example.
- quickstart/rma, simple remote memory access example
- quickstart/quicksort, parallel quicksort implementation.
- gtc, gyrokinetic torodial code.
- bfs, breadth-first-search, example code for a graph application.
- sheneos, partitioning of large data sets.
- accumulator, simple component example.
- balancing/os_thread_num, balancing/px_thread_phase, examples demonstrating load balancing and work stealing.

API changes

- Added `hpx::find_all_localities`.
- Added `hpx::terminate` for non-graceful termination of applications.
- Added `hpx::lcos::async` functions for simpler asynchronous programming.
- Added new AGAS interface for handling of symbolic namespace (`hpx::agas::*`).
- Renamed `hpx::components::wait` to `hpx::lcos::wait`.
- Renamed `hpx::lcos::future_value` to `hpx::lcos::promise`.
- Renamed `hpx::lcos::recursive_mutex` to `hpx::lcos::local_recursive_mutex`, `hpx::lcos::mutex` to `hpx::lcos::local_mutex`
- Removed support for Boost versions older than V1.38, recommended Boost version is now V1.47 and newer.

- Removed `hpx::process` (this will be replaced by a real process implementation in the future).
- Removed non-functional LCO code (`hpx::lcos::dataflow`, `hpx::lcos::thunk`, `hpx::lcos::dataflow_variable`).
- Removed deprecated `hpx::naming::full_address`.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- Issue #28[6418] - Integrate Windows/Linux CMake code for *HPX* core
- Issue #32[6419] - hpx::cout() should be hpx::cout
- Issue #33[6420] - AGAS V2 legacy client does not properly handle error_code
- Issue #60[6421] - AGAS: allow for registerid to optionally take ownership of the gid
- Issue #62[6422] - adaptive1d compilation failure in Fusion
- Issue #64[6423] - Parcel subsystem doesn't resolve domain names
- Issue #83[6424] - No error handling if no console is available
- Issue #84[6425] - No error handling if a hosted locality is treated as the bootstrap server
- Issue #90[6426] - Add general commandline option -N
- Issue #91[6427] - Add possibility to read command line arguments from file
- Issue #92[6428] - Always log exceptions/errors to the log file
- Issue #93[6429] - Log the command line/program name
- Issue #95[6430] - Support for distributed launches
- Issue #97[6431] - Attempt to create a bad component type in AMR examples
- Issue #100[6432] - factorial and factorial_get examples trigger AGAS component type assertions
- Issue #101[6433] - Segfault when hpx::process::here() is called in fibonacci2
- Issue #102[6434] - unknown_component_address in int_object_semaphore_client
- Issue #114[6435] - marduk raises assertion with default parameters

[6418] https://github.com/STEllAR-GROUP/hpx/issues/28
[6419] https://github.com/STEllAR-GROUP/hpx/issues/32
[6420] https://github.com/STEllAR-GROUP/hpx/issues/33
[6421] https://github.com/STEllAR-GROUP/hpx/issues/60
[6422] https://github.com/STEllAR-GROUP/hpx/issues/62
[6423] https://github.com/STEllAR-GROUP/hpx/issues/64
[6424] https://github.com/STEllAR-GROUP/hpx/issues/83
[6425] https://github.com/STEllAR-GROUP/hpx/issues/84
[6426] https://github.com/STEllAR-GROUP/hpx/issues/90
[6427] https://github.com/STEllAR-GROUP/hpx/issues/91
[6428] https://github.com/STEllAR-GROUP/hpx/issues/92
[6429] https://github.com/STEllAR-GROUP/hpx/issues/93
[6430] https://github.com/STEllAR-GROUP/hpx/issues/95
[6431] https://github.com/STEllAR-GROUP/hpx/issues/97
[6432] https://github.com/STEllAR-GROUP/hpx/issues/100
[6433] https://github.com/STEllAR-GROUP/hpx/issues/101
[6434] https://github.com/STEllAR-GROUP/hpx/issues/102
[6435] https://github.com/STEllAR-GROUP/hpx/issues/114

- Issue #115[6436] - Logging messages for SMP runs (on the console) shouldn't be buffered
- Issue #119[6437] - marduk linking strategy breaks other applications
- Issue #121[6438] - pbsdsh problem
- Issue #123[6439] - marduk, dataflow and adaptive1d fail to build
- Issue #124[6440] - Lower default preprocessing arity
- Issue #125[6441] - Move hpx::detail::diagnostic_information out of the detail namespace
- Issue #126[6442] - Test definitions for AGAS reference counting
- Issue #128[6443] - Add averaging performance counter
- Issue #129[6444] - Error with endian.hpp while building adaptive1d
- Issue #130[6445] - Bad initialization of performance counters
- Issue #131[6446] - Add global startup/shutdown functions to component modules
- Issue #132[6447] - Avoid using auto_ptr
- Issue #133[6448] - On Windows hpx.dll doesn't get installed
- Issue #134[6449] - HPX_LIBRARY does not reflect real library name (on Windows)
- Issue #135[6450] - Add detection of unique_ptr to build system
- Issue #137[6451] - Add command line option allowing to repeatedly evaluate performance counters
- Issue #139[6452] - Logging is broken
- Issue #140[6453] - CMake problem on windows
- Issue #141[6454] - Move all non-component libraries into $PREFIX/lib/hpx
- Issue #143[6455] - adaptive1d throws an exception with the default command line options
- Issue #146[6456] - Early exception handling is broken
- Issue #147[6457] - Sheneos doesn't link on Linux
- Issue #149[6458] - sheneos_test hangs

6436 https://github.com/STEllAR-GROUP/hpx/issues/115
6437 https://github.com/STEllAR-GROUP/hpx/issues/119
6438 https://github.com/STEllAR-GROUP/hpx/issues/121
6439 https://github.com/STEllAR-GROUP/hpx/issues/123
6440 https://github.com/STEllAR-GROUP/hpx/issues/124
6441 https://github.com/STEllAR-GROUP/hpx/issues/125
6442 https://github.com/STEllAR-GROUP/hpx/issues/126
6443 https://github.com/STEllAR-GROUP/hpx/issues/128
6444 https://github.com/STEllAR-GROUP/hpx/issues/129
6445 https://github.com/STEllAR-GROUP/hpx/issues/130
6446 https://github.com/STEllAR-GROUP/hpx/issues/131
6447 https://github.com/STEllAR-GROUP/hpx/issues/132
6448 https://github.com/STEllAR-GROUP/hpx/issues/133
6449 https://github.com/STEllAR-GROUP/hpx/issues/134
6450 https://github.com/STEllAR-GROUP/hpx/issues/135
6451 https://github.com/STEllAR-GROUP/hpx/issues/137
6452 https://github.com/STEllAR-GROUP/hpx/issues/139
6453 https://github.com/STEllAR-GROUP/hpx/issues/140
6454 https://github.com/STEllAR-GROUP/hpx/issues/141
6455 https://github.com/STEllAR-GROUP/hpx/issues/143
6456 https://github.com/STEllAR-GROUP/hpx/issues/146
6457 https://github.com/STEllAR-GROUP/hpx/issues/147
6458 https://github.com/STEllAR-GROUP/hpx/issues/149

- Issue #154[6459] - Compilation fails for r5661
- Issue #155[6460] - Sine performance counters example chokes on chrono headers
- Issue #156[6461] - Add build type to –version
- Issue #157[6462] - Extend AGAS caching to store gid ranges
- Issue #158[6463] - r5691 doesn't compile
- Issue #160[6464] - Re-add AGAS function for resolving a locality to its prefix
- Issue #168[6465] - Managed components should be able to access their own GID
- Issue #169[6466] - Rewrite AGAS future pool
- Issue #179[6467] - Complete switch to request class for AGAS server interface
- Issue #182[6468] - Sine performance counter is loaded by other examples
- Issue #185[6469] - Write tests for symbol namespace reference counting
- Issue #191[6470] - Assignment of read-only variable in point_geometry
- Issue #200[6471] - Seg faults when querying performance counters
- Issue #204[6472] - –ifnames and suffix stripping needs to be more generic
- Issue #205[6473] - –list-* and –print-counter-* options do not work together and produce no warning
- Issue #207[6474] - Implement decrement entry merging
- Issue #208[6475] - Replace the spinlocks in AGAS with hpx::lcos::local_mutexes
- Issue #210[6476] - Add an –ifprefix option
- Issue #214[6477] - Performance test for PX-thread creation
- Issue #216[6478] - VS2010 compilation
- Issue #222[6479] - r6045 context_linux_x86.hpp
- Issue #223[6480] - fibonacci hangs when changing the state of an active thread
- Issue #225[6481] - Active threads end up in the FEB wait queue

[6459] https://github.com/STEllAR-GROUP/hpx/issues/154
[6460] https://github.com/STEllAR-GROUP/hpx/issues/155
[6461] https://github.com/STEllAR-GROUP/hpx/issues/156
[6462] https://github.com/STEllAR-GROUP/hpx/issues/157
[6463] https://github.com/STEllAR-GROUP/hpx/issues/158
[6464] https://github.com/STEllAR-GROUP/hpx/issues/160
[6465] https://github.com/STEllAR-GROUP/hpx/issues/168
[6466] https://github.com/STEllAR-GROUP/hpx/issues/169
[6467] https://github.com/STEllAR-GROUP/hpx/issues/179
[6468] https://github.com/STEllAR-GROUP/hpx/issues/182
[6469] https://github.com/STEllAR-GROUP/hpx/issues/185
[6470] https://github.com/STEllAR-GROUP/hpx/issues/191
[6471] https://github.com/STEllAR-GROUP/hpx/issues/200
[6472] https://github.com/STEllAR-GROUP/hpx/issues/204
[6473] https://github.com/STEllAR-GROUP/hpx/issues/205
[6474] https://github.com/STEllAR-GROUP/hpx/issues/207
[6475] https://github.com/STEllAR-GROUP/hpx/issues/208
[6476] https://github.com/STEllAR-GROUP/hpx/issues/210
[6477] https://github.com/STEllAR-GROUP/hpx/issues/214
[6478] https://github.com/STEllAR-GROUP/hpx/issues/216
[6479] https://github.com/STEllAR-GROUP/hpx/issues/222
[6480] https://github.com/STEllAR-GROUP/hpx/issues/223
[6481] https://github.com/STEllAR-GROUP/hpx/issues/225

- Issue #226[6482] - VS Build Error for Accumulator Client
- Issue #228[6483] - Move all traits into namespace hpx::traits
- Issue #229[6484] - Invalid initialization of reference in thread_init_data
- Issue #235[6485] - Invalid GID in iostreams
- Issue #238[6486] - Demangle type names for the default implementation of get_action_name
- Issue #241[6487] - C++11 support breaks GCC 4.5
- Issue #247[6488] - Reference to temporary with GCC 4.4
- Issue #248[6489] - Seg fault at shutdown with GCC 4.4
- Issue #253[6490] - Default component action registration kills compiler
- Issue #272[6491] - G++ unrecognized command line option
- Issue #273[6492] - quicksort example doesn't compile
- Issue #277[6493] - Invalid CMake logic for Windows

2.10.2 Namespace changes

HPX V1.9.0 Namespace changes

The latest release includes amongst others changes in the namespaces so that *HPX* facilities correspond to the C++ Standard Library. The old namespaces are deprecated. Below is a comprehensive list of the namespace changes.

Table 2.183: Namespace changes in V1.9.0

Old namespace	New namespace
`hpx::util::mem_fn`	*hpx::mem_fn*
`hpx::util::invoke`	*hpx::invoke*
`hpx::util::invoke_r`	*hpx::invoke_r*
`hpx::util::invoke_fused`	*hpx::invoke_fused*
`hpx::util::invoke_fused_r`	*hpx::invoke_fused_r*
`hpx::util::unlock_guard`	*hpx::unlock_guard*
`hpx::parallel::v1::reduce_by_key`	*hpx::experimental::reduce_by_key*
`hpx::parallel::v1::sort_by_key`	*hpx::experimental::sort_by_key*
`hpx::parallel::task_canceled_exception`	*hpx::experimental::task_canceled_exception*
`hpx::parallel::task_block`	*hpx::experimental::task_block*
`hpx::parallel::define_task_block`	*hpx::experimental::define_task_block* |
`hpx::parallel::define_task_block_restore_thread`	*hpx::experimental::define_task_block_restore_thread*
`hpx::execution::experimental::task_group`	*hpx::experimental::task_group*

[6482] https://github.com/STEllAR-GROUP/hpx/issues/226
[6483] https://github.com/STEllAR-GROUP/hpx/issues/228
[6484] https://github.com/STEllAR-GROUP/hpx/issues/229
[6485] https://github.com/STEllAR-GROUP/hpx/issues/235
[6486] https://github.com/STEllAR-GROUP/hpx/issues/238
[6487] https://github.com/STEllAR-GROUP/hpx/issues/241
[6488] https://github.com/STEllAR-GROUP/hpx/issues/247
[6489] https://github.com/STEllAR-GROUP/hpx/issues/248
[6490] https://github.com/STEllAR-GROUP/hpx/issues/253
[6491] https://github.com/STEllAR-GROUP/hpx/issues/272
[6492] https://github.com/STEllAR-GROUP/hpx/issues/273
[6493] https://github.com/STEllAR-GROUP/hpx/issues/277

2.11 Citing *HPX*

Please cite *HPX* whenever you use it for publications. Use our paper in The Journal of Open Source Software as the main citation for *HPX*: [6494]. Use the Zenodo entry for referring to the latest version of *HPX*: [6495]. Entries for citing specific versions of *HPX* can also be found at [6496].

2.12 *HPX* users

A list of institutions and projects using *HPX* can be found on the *HPX* Users[6497] page.

2.13 About *HPX*

2.13.1 History

The development of High Performance ParalleX (*HPX*) began in 2007. At that time, Hartmut Kaiser became interested in the work done by the ParalleX group at the Center for Computation and Technology (CCT)[6498], a multi-disciplinary research institute at Louisiana State University (LSU)[6499]. The ParalleX group was working to develop a new and experimental execution model for future high performance computing architectures. This model was christened ParalleX. The first implementations of ParalleX were crude, and many of those designs had to be discarded entirely. However, over time the team learned quite a bit about how to design a parallel, distributed runtime system which implements the concepts of ParalleX.

From the very beginning, this endeavour has been a group effort. In addition to a handful of interested researchers, there have always been graduate and undergraduate students participating in the discussions, design, and implementation of *HPX*. In 2011 we decided to formalize our collective research efforts by creating the STE||AR[6500] group (**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch). Over time, the team grew to include researchers around the country and the world. In 2014, the STE||AR[6501] Group was reorganized to become the international community it is today. This consortium of researchers aims to develop stable, sustainable, and scalable tools which will enable application developers to exploit the parallelism latent in the machines of today and tomorrow. Our goal of the *HPX* project is to create a high quality, freely available, open source implementation of ParalleX concepts for conventional and future systems by building a modular and standards conforming runtime system for SMP and distributed application environments. The API exposed by *HPX* is conformant to the interfaces defined by the C++ ISO Standard and adheres to the programming guidelines used by the Boost[6502] collection of C++ libraries. We steer the development of *HPX* with real world applications and aim to provide a smooth migration path for domain scientists.

To learn more about STE||AR[6503] and ParalleX, see *People* and *Why HPX?*.

[6494] https://joss.theoj.org/papers/022e5917b95517dff20cd3742ab95eca
[6495] https://doi.org/10.5281/zenodo.598202
[6496] https://doi.org/10.5281/zenodo.598202
[6497] https://hpx.stellar-group.org/hpx-users/
[6498] https://www.cct.lsu.edu
[6499] https://www.lsu.edu
[6500] https://stellar-group.org
[6501] https://stellar-group.org
[6502] https://www.boost.org/
[6503] https://stellar-group.org

2.13.2 People

The STE||AR[6504] Group (pronounced as stellar) stands for "**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch". We are an international group of faculty, researchers, and students working at various institutions around the world. The goal of the STE||AR[6505] Group is to promote the development of scalable parallel applications by providing a community for ideas, a framework for collaboration, and a platform for communicating these concepts to the broader community.

Our work is focused on building technologies for scalable parallel applications. *HPX*, our general purpose C++ runtime system for parallel and distributed applications, is no exception. We use *HPX* for a broad range of scientific applications, helping scientists and developers to write code which scales better and shows better performance compared to more conventional programming models such as MPI.

HPX is based on *ParalleX* which is a new (and still experimental) parallel execution model aiming to overcome the limitations imposed by the current hardware and the techniques we use to write applications today. Our group focuses on two types of applications - those requiring excellent strong scaling, allowing for a dramatic reduction of execution time for fixed workloads and those needing highest level of sustained performance through massive parallelism. These applications are presently unable (through conventional practices) to effectively exploit a relatively small number of cores in a multi-core system. By extension, these application will not be able to exploit high-end exascale computing systems which are likely to employ hundreds of millions of such cores by the end of this decade.

Critical bottlenecks to the effective use of new generation high performance computing (HPC) systems include:

- *Starvation*: due to lack of usable application parallelism and means of managing it,
- *Overhead*: reduction to permit strong scalability, improve efficiency, and enable dynamic resource management,
- *Latency*: from remote access across system or to local memories,
- *Contention*: due to multicore chip I/O pins, memory banks, and system interconnects.

The ParalleX model has been devised to address these challenges by enabling a new computing dynamic through the application of message-driven computation in a global address space context with lightweight synchronization. The work on *HPX* is centered around implementing the concepts as defined by the ParalleX model. *HPX* is currently targeted at conventional machines, such as classical Linux based Beowulf clusters and SMP nodes.

We fully understand that the success of *HPX* (and ParalleX) is very much the result of the work of many people. To see a list of who is contributing see our tables below.

[6504] https://stellar-group.org
[6505] https://stellar-group.org

HPX contributors

Table 2.184: Contributors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT)[6506], Louisiana State University (LSU)[6507]	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture[6508], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[6509]	thom.heller@gmail.com
Agustin Berge		agustinberge@gmail.com
Mikael Simberg	Swiss National Supercomputing Centre[6510]	simbergm@cscs.ch
John Biddiscombe	Swiss National Supercomputing Centre[6511]	biddisco@cscs.ch
Anton Bikineev	Center for Computation and Technology (CCT)[6512], Louisiana State University (LSU)[6513]	ant.bikineev@gmail.com
Martin Stumpf	Department of Computer Science 3 - Computer Architecture[6514], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[6515]	martin.h.stumpf@gmail.com
Bryce Adelstein Lelbach		brycelelbach@gmail.com
Shuangyang Yang	Center for Computation and Technology (CCT)[6516], Louisiana State University (LSU)[6517]	syang16@cct.lsu.edu
Jeroen Habraken		vexocide@gmail.com
Steven Brandt	Center for Computation and Technology (CCT)[6518], Louisiana State University (LSU)[6519]	sbrandt@cct.lsu.edu
Antoine Tran Tan	Paris-Saclay University[6520],	antoine.trantan@universite-paris-saclay.fr
Adrian S. Lemoine	AMD[6521]	Adrian.Lemoine@amd.com
Maciej Brodowicz		maciekab@gmail.com
Giannis Gonidelis	Center for Computation and Technology (CCT)[6522], Louisiana State University (LSU)[6523]	gonidelis@hotmail.com

6506 https://www.cct.lsu.edu
6507 https://www.lsu.edu
6508 https://www3.cs.fau.de
6509 https://www.fau.de
6510 https://www.cscs.ch
6511 https://www.cscs.ch
6512 https://www.cct.lsu.edu
6513 https://www.lsu.edu
6514 https://www3.cs.fau.de
6515 https://www.fau.de
6516 https://www.cct.lsu.edu
6517 https://www.lsu.edu
6518 https://www.cct.lsu.edu
6519 https://www.lsu.edu
6520 https://www.universite-paris-saclay.fr/en
6521 https://www.amd.com/en
6522 https://www.cct.lsu.edu
6523 https://www.lsu.edu

Contributors to this document

Table 2.185: Documentation authors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT)[6524], Louisiana State University (LSU)[6525]	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture[6526], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[6527]	thom.heller@gmail.com
Bryce Adelstein Lelbach		brycelelbach@gmail.com
Vinay C Amatya	Center for Computation and Technology (CCT)[6528], Louisiana State University (LSU)[6529]	vamatya@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT)[6530], Louisiana State University (LSU)[6531]	sbrandt@cct.lsu.edu
Maciej Brodowicz		maciekab@gmail.com
Adrian S. Lemoine	AMD[6532]	Adrian.Lemoine@amd.com
Rebecca Stobaugh		rstobaugh1@gmail.com
Dimitra Karatza	Faculty of Electrical Engineering, Mathematics & Computer Science[6533], Delft University of Technology[6534]	dimitra.karatza11@gmail.com
Bhumit Attarde		bhumitattarde2@gmail.com

Acknowledgements

Thanks also to the following people who contributed directly or indirectly to the project through discussions, pull requests, documentation patches, etc.

- Panos Syskakis for benchmarking and optimizing our parallel algorithms.
- Shreyas Atre, for contributing fixes to our implementation of senders/receivers and extending our coroutines integration with senders/receivers.
- Alexander Neumann, for contributing fixes to the cmake build system.
- Dimitra Karatza, for her work on refactoring the documentation and providing a new user-friendly environment during and after Google Season of Docs 2021.

[6524] https://www.cct.lsu.edu
[6525] https://www.lsu.edu
[6526] https://www3.cs.fau.de
[6527] https://www.fau.de
[6528] https://www.cct.lsu.edu
[6529] https://www.lsu.edu
[6530] https://www.cct.lsu.edu
[6531] https://www.lsu.edu
[6532] https://www.amd.com/en
[6533] https://www.tudelft.nl/en/eemcs
[6534] https://www.tudelft.nl/en/

- Srinivas Yadav, for his work on SIMD support in algorithms before and during Google Summer of Code 2021.
- Akhil Nair, for his work on adapting algorithms to C++20 before and during Google Summer of Code 2021.
- Alexander Toktarev, for updating the parallel algorithm customization points to use `tag_fallback_invoke` for the default implementations.
- Brice Goglin, for reporting and helping fix issues related to the integration of hwloc in *HPX*.
- Giannis Gonidelis, for his work on the ranges adaptation during the Google Summer of Code 2020.
- Auriane Reverdell (Swiss National Supercomputing Centre[6535]), for her tireless work on refactoring our CMake setup and modularizing *HPX*.
- Christopher Hinz, for his work on refactoring our CMake setup.
- Weile Wei, for fixing *HPX* builds with CUDA on Summit.
- Severin Strobl, for fixing our CMake setup related to linking and adding new entry points to the *HPX* runtime.
- Rebecca Stobaugh, for her major documentation review and contributions during and after the 2019 Google Season of Documentation.
- Jan Melech, for adding automatic serialization of simple structs.
- Austin McCartney, for adding concept emulation of the Ranges TS bidirectional and random access iterator concepts.
- Marco Diers, reporting and fixing issues related PMIx.
- Maximilian Bremer, for reporting multiple issues and extending the component migration tests.
- Piotr Mikolajczyk, for his improvements and fixes to the set and count algorithms.
- Grant Rostig, for reporting several deficiencies on our web pages.
- Jakub Golinowski, for implementing an *HPX* backend for OpenCV and in the process improving documentation and reporting issues.
- Mikael Simberg (Swiss National Supercomputing Centre[6536]), for his tireless help cleaning up and maintaining *HPX*.
- Tianyi Zhang, for his work on HPXMP.
- Shahrzad Shirzad, for her contributions related to Phylanx.
- Christopher Ogle, for his contributions to the parallel algorithms.
- Surya Priy, for his work with statistic performance counters.
- Anushi Maheshwari, for her work on random number generation.
- Bruno Pitrus, for his work with parallel algorithms.
- Nikunj Gupta, for rewriting the implementation of `hpx_main.hpp` and for his fixes for tests.
- Christopher Taylor, for his interest in *HPX* and the fixes he provided. Chris also contributed support for RISC-V architectures.
- Shoshana Jakobovits, for her work on the resource partitioner.
- Denis Blank, who re-wrote our unwrapped function to accept plain values arbitrary containers, and properly deal with nested futures.
- Ajai V. George, who implemented several of the parallel algorithms.

[6535] https://www.cscs.ch
[6536] https://www.cscs.ch

- Taeguk Kwon, who worked on implementing parallel algorithms as well as adapting the parallel algorithms to the Ranges TS.
- Zach Byerly (Louisiana State University (LSU)[6537]), who in his work developing applications on top of *HPX* opened tickets and contributed to the *HPX* examples.
- Daniel Estermann, for his work porting *HPX* to the Raspberry Pi.
- Alireza Kheirkhahan (Louisiana State University (LSU)[6538]), who built and administered our local cluster as well as his work in distributed IO.
- Abhimanyu Rawat, who worked on stack overflow detection.
- David Pfander, who improved signal handling in *HPX*, provided his optimization expertise, and worked on incorporating the Vc vectorization into *HPX*.
- Denis Demidov, who contributed his insights with VexCL.
- Khalid Hasanov, who contributed changes which allowed to run *HPX* on 64Bit power-pc architectures.
- Zahra Khatami (Louisiana State University (LSU)[6539]), who contributed the prefetching iterators and the persistent auto chunking executor parameters implementation.
- Marcin Copik, who worked on implementing GPU support for C++AMP and HCC. He also worked on implementing a HCC backend for *HPX.Compute*.
- Minh-Khanh Do, who contributed the implementation of several segmented algorithms.
- Bibek Wagle (Louisiana State University (LSU)[6540]), who worked on fixing and analyzing the performance of the *parcel* coalescing plugin in *HPX*.
- Lukas Troska, who reported several problems and contributed various test cases allowing to reproduce the corresponding issues.
- Andreas Schaefer, who worked on integrating his library (LibGeoDecomp[6541]) with *HPX*. He reported various problems and submitted several patches to fix issues allowing for a better integration with LibGeoDecomp[6542].
- Satyaki Upadhyay, who contributed several examples to *HPX*.
- Brandon Cordes, who contributed several improvements to the inspect tool.
- Harris Brakmic, who contributed an extensive build system description for building *HPX* with Visual Studio.
- Parsa Amini (Louisiana State University (LSU)[6543]), who refactored and simplified the implementation of *AGAS* in *HPX* and who works on its implementation and optimization.
- Luis Martinez de Bartolome who implemented a build system extension for *HPX* integrating it with the Conan[6544] C/C++ package manager.
- Vinay C Amatya (Louisiana State University (LSU)[6545]), who contributed to the documentation and provided some of the *HPX* examples.
- Kevin Huck and Nick Chaimov (University of Oregon[6546]), who contributed the integration of APEX (Autonomic Performance Environment for eXascale) with *HPX*.
- Francisco Jose Tapia, who helped with implementing the parallel sort algorithm for *HPX*.

[6537] https://www.lsu.edu
[6538] https://www.lsu.edu
[6539] https://www.lsu.edu
[6540] https://www.lsu.edu
[6541] https://www.libgeodecomp.org/
[6542] https://www.libgeodecomp.org/
[6543] https://www.lsu.edu
[6544] https://www.conan.io/
[6545] https://www.lsu.edu
[6546] https://uoregon.edu/

- Patrick Diehl, who worked on implementing CUDA support for our companion library targeting GPGPUs (HPXCL[6547]).
- Eric Lemanissier contributed fixes to allow compilation using the MingW toolchain.
- Nidhi Makhijani who helped cleaning up some enum consistencies in *HPX* and contributed to the resource manager used in the thread scheduling subsystem. She also worked on *HPX* in the context of the Google Summer of Code 2015.
- Larry Xiao, Devang Bacharwar, Marcin Copik, and Konstantin Kronfeldner who worked on *HPX* in the context of the Google Summer of Code program 2015.
- Daniel Bourgeois (Center for Computation and Technology (CCT)[6548]) who contributed to *HPX* the implementation of several parallel algorithms (as proposed by N4313[6549]).
- Anuj Sharma and Christopher Bross (Department of Computer Science 3 - Computer Architecture[6550]), who worked on *HPX* in the context of the Google Summer of Code[6551] program 2014.
- Martin Stumpf (Department of Computer Science 3 - Computer Architecture[6552]), who rebuilt our contiguous testing infrastructure (see the *HPX* Buildbot Website[6553]). Martin is also working on HPXCL[6554] (mainly all work related to OpenCL[6555]) and implementing an *HPX* backend for POCL[6556], a portable computing language solution based on OpenCL[6557].
- Grant Mercer (University of Nevada, Las Vegas[6558]), who helped creating many of the parallel algorithms (as proposed by N4313[6559]).
- Damond Howard (Louisiana State University (LSU)[6560]), who works on HPXCL[6561] (mainly all work related to CUDA[6562]).
- Christoph Junghans (Los Alamos National Lab), who helped making our buildsystem more portable.
- Antoine Tran Tan (Laboratoire de Recherche en Informatique, Paris), who worked on integrating *HPX* as a backend for NT2[6563]. He also contributed an implementation of an API similar to Fortran co-arrays on top of *HPX*.
- John Biddiscombe (Swiss National Supercomputing Centre[6564]), who helped with the BlueGene/Q port of *HPX*, implemented the parallel sort algorithm, and made several other contributions.
- Erik Schnetter (Perimeter Institute for Theoretical Physics), who greatly helped to make *HPX* more robust by submitting a large amount of problem reports, feature requests, and made several direct contributions.
- Mathias Gaunard (Metascale), who contributed several patches to reduce compile time warnings generated while compiling *HPX*.
- Andreas Buhr, who helped with improving our documentation, especially by suggesting some fixes for inconsistencies.

[6547] https://github.com/STEllAR-GROUP/hpxcl/
[6548] https://www.cct.lsu.edu
[6549] http://wg21.link/n4313
[6550] https://www3.cs.fau.de
[6551] https://developers.google.com/open-source/soc/
[6552] https://www3.cs.fau.de
[6553] http://rostam.cct.lsu.edu/
[6554] https://github.com/STEllAR-GROUP/hpxcl/
[6555] https://www.khronos.org/opencl/
[6556] https://portablecl.org/
[6557] https://www.khronos.org/opencl/
[6558] https://www.unlv.edu
[6559] http://wg21.link/n4313
[6560] https://www.lsu.edu
[6561] https://github.com/STEllAR-GROUP/hpxcl/
[6562] https://www.nvidia.com/object/cuda_home_new.html
[6563] https://www.numscale.com/nt2/
[6564] https://www.cscs.ch

- Patricia Grubel (New Mexico State University[6565]), who contributed the description of the different *HPX* thread scheduler policies and is working on the performance analysis of our thread scheduling subsystem.
- Lars Viklund, whose wit, passion for testing, and love of odd architectures has been an amazing contribution to our team. He has also contributed platform specific patches for FreeBSD and MSVC12.
- Agustin Berge, who contributed patches fixing some very nasty hidden template meta-programming issues. He rewrote large parts of the API elements ensuring strict conformance with the C++ ISO Standard.
- Anton Bikineev for contributing changes to make using `boost::lexical_cast` safer, he also contributed a thread safety fix to the iostreams module. He also contributed a complete rewrite of the serialization infrastructure replacing Boost.Serialization inside *HPX*.
- Pyry Jahkola, who contributed the Mac OS build system and build documentation on how to build *HPX* using Clang and libc++.
- Mario Mulansky, who created an *HPX* backend for his Boost.Odeint library, and who submitted several test cases allowing us to reproduce and fix problems in *HPX*.
- Rekha Raj, who contributed changes to the description of the Windows build instructions.
- Jeremy Kemp how worked on an *HPX* OpenMP backend and added regression tests.
- Alex Nagelberg for his work on implementing a C wrapper API for *HPX*.
- Chen Guo, helvihartmann, Nicholas Pezolano, and John West who added and improved examples in *HPX*.
- Joseph Kleinhenz, Markus Elfring, Kirill Kropivyansky, Alexander Neundorf, Bryant Lam, and Alex Hirsch who improved our CMake.
- Tapasweni Pathak, Praveen Velliengiri, Jean-Loup Tastet, Michael Levine, Aalekh Nigam, HadrienG2, Prayag Verma, lslada, Alex Myczko, and Avyav Kumar who improved the documentation.
- Jayesh Badwaik, J. F. Bastien, Christoph Garth, Christopher Hinz, Brandon Kohn, Mario Lang, Maikel Nadolski, pierrele, hendrx, Dekken, woodmeister123, xaguilar, Andrew Kemp, Dylan Stark, Matthew Anderson, Jeremy Wilke, Jiazheng Yuan, CyberDrudge, david8dixon, Maxwell Reeser, Raffaele Solca, Marco Ippolito, Jules Penuchot, Weile Wei, Severin Strobl, Kor de Jong, albestro, Jeff Trull, Yuri Victorovich, and Gregor Daiß who contributed to the general improvement of *HPX*.

HPX Funding Acknowledgements[6566] lists current and past funding sources for *HPX*. Special thanks to Google Summer of Code[6567] and Google Season of Docs[6568] for the continuous support they provide which helps us enhance both our code and our documentation.

[6565] https://www.nmsu.edu
[6566] https://hpx.stellar-group.org/funding-acknowledgements/
[6567] https://developers.google.com/open-source/soc/
[6568] https://developers.google.com/season-of-docs

CHAPTER

THREE

INDEX

- genindex

INDEX

Symbols

A

B

C

D

F

H

L

P

S