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.

If you can’t find what you’re looking for in the documentation, please:

- open an issue on GitHub¹;
- contact us on IRC, the HPX channel on the C++ Slack², or on our mailing list³; or
- read or ask questions tagged with HPX on StackOverflow⁴.

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

¹ https://github.com/STEllAR-GROUP/hpx/issues
² https://cpplang.slack.com
³ hpx-users@stellar.cct.lsu.edu
⁴ https://stackoverflow.com/questions/tagged/hpx
**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 STEllAR 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.
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.

2.1.1 Installing HPX

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

1. **vc pkg**
   
   You can download and install HPX using the vc pkg\(^5\) dependency manager:

   ```sh
   $ vc pkg install hpx
   ```

2. **Spack**

   Another way to install HPX is using Spack\(^6\):

   ```sh
   $ spack install hpx
   ```

\(^5\) [https://github.com/Microsoft/vcpkg](https://github.com/Microsoft/vcpkg)

3. Fedora

Installation can be done with Fedora\(^7\) as well:

```
$ dnf install hpx*
```

4. Arch Linux

\textit{HPX} is available in the Arch User Repository (AUR)\(^8\) as hpx too.

More information or alternatives regarding the installation can be found in the \textit{HPX build system}, a detailed guide with thorough explanation of ways to build and use \textit{HPX}.

### 2.1.2 Hello, World!

To get started with this minimal example you need to create a new project directory and a file \texttt{CMakeLists.txt} with the contents below in order to build an executable using CMake and \textit{HPX}:

```
cmake_minimum_required(VERSION 3.18)
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 \texttt{main.cpp} with the contents below:

```
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

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

Then, in your project directory run the following:

```
$ mkdir build && cd build
$ cmake -DCMAKE_PREFIX_PATH=/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 \texttt{hpx::cout}.

- When you include \texttt{hpx\_main\_hpp} \textit{HPX} makes sure that \texttt{main} actually gets launched on the \textit{HPX} runtime. So while it looks almost the same you can now use futures, \texttt{async}, parallel algorithms and more which make use of the \textit{HPX} runtime with lightweight threads.

- \texttt{hpx::cout} is a replacement for \texttt{std::cout} to make sure printing never blocks a lightweight thread. You can read more about \texttt{hpx::cout} in \textit{The HPX I/O-streams component}.

\(^7\) \url{https://fedoraproject.org/wiki/DNF}

\(^8\) \url{https://wiki.archlinux.org/title/Arch_User_Repository}
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: 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 HPX applications).

If you are already familiar with async and futures 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/local/algorithm.hpp>
#include <hpx/local/future.hpp>
#include <hpx/local/init.hpp>
```

(continues on next page)
#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,
            hpx::execution::task),
        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;
}
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.
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. ?? 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.](image)

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 HPX build system for information on configuring and building HPX) and enter:

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

To run the program type:
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 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 for more details). `hpx::init` calls `hpx_main()` after setting up `HPX`, which is where the logic of our program is encoded.

```c
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::local::init_params init_args;
    init_args.desc_cmdline = desc_commandline;

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

---

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

```cpp
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;
    std::uint64_t r = fibonacci(n);

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

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

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

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

    // Invoking the Fibonacci algorithm twice is inefficient.
    // However, we intentionally demonstrate it this way to create some
    // heavy workload.
    hpx::future<std::uint64_t> n1 = hpx::async(fibonacci, n - 1);
    hpx::future<std::uint64_t> n2 = hpx::async(fibonacci, n - 2);

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

This block of code looks similar to regular C++ code. First, \texttt{if (n < 2)}, meaning \texttt{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 \texttt{n} is larger than 1 we spawn two new tasks whose results are contained in \texttt{n1} and \texttt{n2}. This is done using \texttt{hpx::async} which takes as arguments a function (function pointer, object or lambda) and the arguments to the function. Instead of returning a \texttt{std::uint64_t} like \texttt{fibonacci} does, \texttt{hpx::async} returns a future of a \texttt{std::uint64_t}, i.e. \texttt{hpx::future<std::uint64_t>}. Each of these futures represents an asynchronous, recursive call to \texttt{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 \texttt{get} method. The recursive call tree will continue until \texttt{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 \texttt{n}-th value of the Fibonacci sequence.

Note that calling \texttt{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. \texttt{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 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 HPX build system 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:

```cpp
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"
    );
}
```

(continues on next page)
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}
                     elapsed time: {3} [s]"
                     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 actions wrap simple global functions which are not associated with any particular object (we will discuss other types of actions 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():

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

    // We restrict ourselves to execute the Fibonacci function locally.
    hpx::naming::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.

2.2.3 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 HPX build system for information on configuring and building HPX) and enter:

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

To run the program type:

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

2.2. Examples
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()`:

```cpp
// 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::naming::id_type> localities = hpx::find_all_localities();

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

    for (hpx::naming::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::lcos::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:

```cpp
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);

    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::lcos::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, which we can query to determine if the task has
            // completed. We give the task a hint to run on a particular worker
            // thread, but no guarantees are given by the scheduler that the
            // task will actually run on that worker thread.
            hpx::execution::parallel_executor exec(
                hpx::threads::thread_schedule_hint(
                    hpx::threads::thread_schedule_hint_mode::thread,
                    static_cast<std::int16_t>(worker)));
            futures.push_back(hpx::async(exec, hello_world_worker, worker));
        }

        // Wait for all of the futures to finish. The callback version of the
        // hpx::lcos::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::lcos::wait_each doesn't return until
        // all the futures in the vector have returned.
        hpx::lcos::local::spinlock mtx;
        hpx::lcos::wait_each(hpx::unwrapping([&](std::size_t t) {
```
if (std::size_t(-1) != t)
{
    std::lock_guard<hpx::lcos::local::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::lcos::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::lcos::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.

```cpp
def 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.

### 2.2.4 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 HPX build system 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
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:

```cpp
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:

```cpp
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:

```cpp
argument_type value_;  
```

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

```cpp
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:

```cpp
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:

```cpp
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

```cpp
/////////////////////////////////////////////////////////////////////
// Add factory registration functionality.
HPX_REGISTER_COMPONENT_MODULE()  
(continues on next page)
```
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:

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

and to invoke component actions:

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

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

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

For readability, we typedef the base class like so:

```cpp
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::apply` to invoke an action in a non-blocking fashion.

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

typedef server::accumulator::reset_action action_type;
    hpx::apply<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:

```cpp
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:

```cpp
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.

### 2.2.5 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 HPX build system 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.

```cpp
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::naming::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);
    // }
    //
    // 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:

```cpp
// 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:

```cpp
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.

### 2.2.6 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. ?? below.

![Fig. 2.2: Heat diffusion example program flow.](image)

We parallelize this code over the following eight examples:

- Example 1
The first example is straight serial code. In this code we instantiate a vector $\mathbf{U}$ that contains two vectors of doubles as seen in the structure `stepper`.

```cpp
struct stepper
{
    // Our partition type
typedef 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
    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.

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

        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)]);
            }
        }
    }
};
```

(continues on next page)
// 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]);
};

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 grain size 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 is 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.

```cpp
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:

```cpp
// 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);
            }
        });
```
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.


2.3 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 HPX build system

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>
<thead>
<tr>
<th>Name</th>
<th>Minimum Version</th>
<th>Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>2.6</td>
<td>x86-32, x86-64, k10m</td>
</tr>
<tr>
<td>BlueGeneQ</td>
<td>V1R2M0</td>
<td>PowerPC A2</td>
</tr>
<tr>
<td>Windows</td>
<td>Any Windows system</td>
<td>x86-32, x86-64</td>
</tr>
<tr>
<td>Mac OSX</td>
<td>Any OSX system</td>
<td>x86-64</td>
</tr>
</tbody>
</table>

Supported compilers

The table below shows the supported compilers for HPX.

<table>
<thead>
<tr>
<th>Name</th>
<th>Minimum Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU Compiler Collection (g++)</td>
<td>7.0</td>
</tr>
<tr>
<td>clang: a C language family frontend for LLVM</td>
<td>7.0</td>
</tr>
<tr>
<td>Visual C++ (x64)</td>
<td>2015</td>
</tr>
</tbody>
</table>

Software and libraries

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Minimum Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build System</td>
<td></td>
</tr>
<tr>
<td>CMake</td>
<td>3.18</td>
</tr>
<tr>
<td>Required Libraries</td>
<td></td>
</tr>
<tr>
<td>Boost</td>
<td>1.71.0</td>
</tr>
<tr>
<td>Portable Hardware Locality (HWLOC)</td>
<td>1.5</td>
</tr>
<tr>
<td>Asio</td>
<td>1.12.0</td>
</tr>
</tbody>
</table>

---

11 https://gcc.gnu.org
12 https://clang.llvm.org/
The most important dependencies are Boost\textsuperscript{18} and Portable Hardware Locality (HWLOC)\textsuperscript{19}. The installation of Boost is described in detail in Boost’s Getting Started\textsuperscript{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\textsuperscript{21}.

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

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

\textbf{Note:} In most configurations, \textit{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 \textit{HPX}, you will find that \textit{HPX} may depend on several other libraries such as those listed below.

\textbf{Note:} In order to use a high speed parcelport, we currently recommend configuring \textit{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.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Name & Minimum version \\
\hline
google-perftools\textsuperscript{22} & 1.7.1 \\
jemalloc\textsuperscript{23} & 2.1.0 \\
mimalloc\textsuperscript{24} & 1.0.0 \\
Performance Application Programming Interface (PAPI) & \\
\hline
\end{tabular}
\caption{Optional software prerequisites for \textit{HPX}}
\end{table}

\textbf{Getting HPX}

Download a tarball of the latest release from \textit{HPX Downloads}\textsuperscript{25} and unpack it or clone the repository directly using git:

\begin{verbatim}
$ git clone https://github.com/STEllAR-GROUP/hpx.git
\end{verbatim}

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

\begin{verbatim}
$ git clone https://github.com/STEllAR-GROUP/hpx.git
\end{verbatim}

\textsuperscript{14} https://www.cmake.org
\textsuperscript{15} https://www.boost.org/
\textsuperscript{16} https://www.open-mpi.org/projects/hwloc/
\textsuperscript{17} https://think-async.com/Asio/
\textsuperscript{18} https://www.boost.org/
\textsuperscript{19} https://www.open-mpi.org/projects/hwloc/
\textsuperscript{20} https://www.boost.org/more/getting_started/index.html
\textsuperscript{21} https://www.open-mpi.org/software/hwloc/v1.11
\textsuperscript{22} https://code.google.com/p/gperftools
\textsuperscript{23} http://jemalloc.net
\textsuperscript{24} http://microsoft.github.io/mimalloc/
\textsuperscript{25} https://hpx.stellar-group.org/downloads/
Building HPX

Basic information

The build system for HPX is based on CMake, 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.

Once CMake has been run, the build process can be started. The HPX build process is highly configurable through CMake, and various CMake variables influence the build process. 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.

For a complete list of available CMake variables that influence the build of HPX, see CMake variables used to configure HPX.

The variables can be used to refine the recipes that can be found at Platform specific build recipes which show 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).

As HPX is a modern C++ library which relies on C++17 by default. The use of more recent standards can be opted into explicitly. If you want to force HPX to use a specific C++ standard version, you can use the following CMake variables:

- HPX_WITH_CXX17: [ Deprecated] C++17 is now the default C++ standard used in HPX.
- HPX_WITH_CXX20: [ Deprecated] In order to use the C++20 standard, it is preferable to set CMAKE_CXX_STANDARD and HPX_USE_CMAKE_CXX_STANDARD to ON.

---

26 https://www.cmake.org
27 https://www.cmake.org/cmake/resources/software.html
Build types

CMake 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 variable used to set the build type is \texttt{CMAKE\_BUILD\_TYPE} (for more information see the CMake Documentation\(^{28}\)). 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 \textit{HPX} API, the C++ Macro \texttt{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.

\textbf{Important:} We currently don’t guarantee ABI compatibility between Debug and Release builds. Please make sure that applications built against \textit{HPX} use the same build type as you used to build \textit{HPX}. For CMake builds, this means that the \texttt{CMAKE\_BUILD\_TYPE} variables have to match and for projects not using CMake\(^{29}\), the \texttt{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, 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 \textit{HPX} libraries and examples, and install them to your chosen location (recommended):

     $ cmake -DCMAKE\_INSTALL\_PREFIX=/install/path ..

   \textbf{Tip:} If you want to change CMake 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 \texttt{Release} and \texttt{Debug} modes.

   - To install \textit{HPX} to the default system folders, simply leave out the \texttt{CMAKE\_INSTALL\_PREFIX} option:

     $ cmake ..

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

\(^{28}\) https://cmake.org/cmake/help/latest/variable/CMAKE\_BUILD\_TYPE.html
\(^{29}\) https://www.cmake.org
$ 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:

$ 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 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

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

• Download the hwloc V1.11.0 (or the latest version) from here and unpack it.

• Download the latest Boost libraries from here 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:

  30 https://blog.kitware.com/cmake-3-18-1-available-for-download/
  31 http://www.open-mpi.org/software/hwloc/v1.11/downloads/hwloc-win64-build-1.11.0.zip
  32 https://www.boost.org/users/download/
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:

![CMake GUI with example settings](image)

Fig. 2.3: Example CMake path settings.

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 three new environment variables (in CMake, not in Windows environment): BOOST_ROOT, HWLOC_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.
  - **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:

![Example CMake adding entry.](image)

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:

Tests and examples

Running tests

To build the tests:

```bash
$ cmake --build . --target tests
```

To control which tests to run use ctest:

- To run single tests, for example a test for `for_loop`:

```bash
$ ctest --output-on-failure -R tests.unit.modules.algorithms.for_loop
```

- To run a whole group of tests:
Running examples

- To build (and install) all examples invoke:

```bash
$ cmake -DHPX_WITH_EXAMPLES=On .
$ make examples
$ make install
```

- To build the `hello_world_1` example run:

```bash
$ 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:

```bash
$ ./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 all available targets. For example, `make help | grep hello_world` outputs the following:
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_ASYNC_CUDA:BOOL`
- `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_COMPRESSOR_WARNINGS_AS_ERRORS:BOOL`
- `HPX_WITH_COMPRESSOR_BZIP2:BOOL`
- `HPX_WITH_COMPRESSOR_SNAPPY:BOOL`
- `HPX_WITH_COMPRESSOR_ZLIB:BOOL`
- `HPX_WITH_COMPUTE_CUDA:BOOL`
- `HPX_WITH_CUDA:BOOL`
- `HPX_WITH_DATAPAR:BOOL`
- `HPX_WITH_DATAPAR_VC:BOOL`
- `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`
- \texttt{HPX\_WITH\_HIP:BOOL}
- \texttt{HPX\_WITH\_LOGGING:BOOL}
- \texttt{HPX\_WITH\_MALLOC:STRING}
- \texttt{HPX\_WITH\_NICE\_THREADLEVEL:BOOL}
- \texttt{HPX\_WITH\_PARCEL\_COALESCING:BOOL}
- \texttt{HPX\_WITH\_PKGCONFIG:BOOL}
- \texttt{HPX\_WITH\_PRECOMPILED\_HEADERS:BOOL}
- \texttt{HPX\_WITH\_RUN\_MAIN\_EVERYWHERE:BOOL}
- \texttt{HPX\_WITH\_STACKOVERFLOW\_DETECTION:BOOL}
- \texttt{HPX\_WITH\_STATIC\_LINKING:BOOL}
- \texttt{HPX\_WITH\_UNITY\_BUILD:BOOL}
- \texttt{HPX\_WITH\_VIM\_YCM:BOOL}
- \texttt{HPX\_WITH\_ZERO\_COPY\_SERIALIZATION\_THRESHOLD:STRING}

\textbf{HPX\_WITH\_ASYNC\_CUDA:BOOL}

ON

\textbf{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)

\textbf{HPX\_WITH\_BENCHMARK\_SCRIPTS\_PATH:PATH}

Directory to place batch scripts in

\textbf{HPX\_WITH\_BUILD\_BINARY\_PACKAGE:BOOL}

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

\textbf{HPX\_WITH\_CHECK\_MODULE\_DEPENDENCIES:BOOL}

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

\textbf{HPX\_WITH\_COMPILER\_WARNINGS:BOOL}

Enable compiler warnings (default: ON)

\textbf{HPX\_WITH\_COMPILER\_WARNINGS\_AS\_ERRORS:BOOL}

Turn compiler warnings into errors (default: OFF)

\textbf{HPX\_WITH\_COMPRESSION\_BZIP2:BOOL}

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

\textbf{HPX\_WITH\_COMPRESSION\_SNAPPY:BOOL}

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

\textbf{HPX\_WITH\_COMPRESSION\_ZLIB:BOOL}

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

\textbf{HPX\_WITH\_COMPUTE\_CUDA:BOOL}

Enable HPX CUDA/HIP compute capability (parallel algorithms) module (default: ON, dependent on HPX\_WITH\_CUDA or HPX\_WITH\_HIP, and HPX\_WITH\_ASYNC\_CUDA) - note: enabling this also enables CUDA/HIP futures via HPX\_WITH\_ASYNC\_CUDA

\textbf{HPX\_WITH\_CUDA:BOOL}

Enable HPX\_WITH\_ASYNC\_CUDA (CUDA or HIP futures) and HPX\_WITH\_COMPUTE\_CUDA (CUDA/HIP enabled parallel algorithms) (default: OFF)

2.3. Manual
HPX Documentation, master

HPX_WITH_DATAPAR:BOOL
Enable data parallel algorithm support (default: ON)

HPX_WITH_DATAPAR_VC:BOOL
Enable data parallel algorithm support using the external Vc library (default: OFF)

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_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_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_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: 128)

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

HPX WITH IO_COUNTERS:BOOL
Enable IO counters (default: ON)

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`
  Emulate SwapContext API for coroutines (Windows only, 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: 64)

- `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 Documentation, master

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)

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`  
  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_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`  
  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_DEBUG_LOG:BOOL`  
- `HPX_WITH_TESTS_DEBUG_LOG_DESTINATION:STRING`  
- `HPX_WITH_TESTS_MAX_THREADS_PER_LOCALITY:STRING`
• 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, implicitly 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)

• HPX_WITH_VERIFY_LOCKS_GLOBALY:BOOL
  Enable global lock verification code (default: OFF, implicitly enabled in debug builds)

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_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, implicitly 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)

HPX_WITH_VERIFY_LOCKS_GLOBALY:BOOL
  Enable global lock verification code (default: OFF, implicitly enabled in debug builds)
Modules options

- **HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE**:BOOL
  Enable compatibility of hpx::tuple with std::tuple. (default: ON)

- **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.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: ON)

- **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)

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.

### 2.3.2 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.

```cpp
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

int main()
{
    // Say hello to the world!
    hpx::cout << "Hello World!\n" << hpx::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 -lhpixiostreams you will need to specify -lhpixiostreamsd.

**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**

```cpp
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include "hello_world_component.hpp"
#include <iostream>
namespace examples {
namespace server {
    void hello_world::invoke()
    {
        hpx::cout << "Hello HPX World!" << std::endl;
    }
}
}}
HPX_REGISTER_COMPONENT_MODULE()

namespace examples { namespace server
{
    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**

```cpp
#pragma once
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
```

(continues on next page)
#include <hpx/hpx.hpp>
#include <hpx/include/actions.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/include/components.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);
    }
}
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::naming::id_type> && f) 
            : base_type(std::move(f))
        {}
        hello_world(hpx::naming::id_type && f) 
            : base_type(std::move(f))
        {}
        void invoke()
        {
            hpx::async<server::hello_world::invoke_action>(this->get_id()).get();
        }
    } 
}

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()
{ 
    
}
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):

```bash
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`:

```bash
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_iostreams_d`.

**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:

```bash
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:

```bash
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` 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:

```cmake
# 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.

---

33 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 \texttt{HPX::hpx} CMake target:

\begin{verbatim}
target_link_libraries(hello_world_component PUBLIC HPX::hpx)
\end{verbatim}

This requires that you have already created the target like this:

\begin{verbatim}
add_library(hello_world_component SHARED hello_world_component.cpp)
target_include_directories(hello_world_component PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
\end{verbatim}

When you link your library to the \texttt{HPX::hpx} CMake target, you will be able use HPX functionality in your library. To use \texttt{main()} as the implicit entry point in your application you must additionally link your application to the CMake target \texttt{HPX::wrap_main}. This target is automatically linked to executables if you are using the macros described below (\textit{Using macros to create new targets}). See \textit{Re-use the main() function as the main HPX entry point} for more information on implicitly using \texttt{main()} as the entry point.

Creating a component requires setting two additional compile definitions:

\begin{verbatim}
target_compile_options(hello_world_component
  HPX_COMPONENT_NAME=hello_world
  HPX_COMPONENT_EXPORTS)
\end{verbatim}

Instead of setting these definitions manually you may link to the \texttt{HPX::component} target, which sets \texttt{HPX_COMPONENT_NAME} to \texttt{hpx_<target_name>}, where \texttt{<target_name>} is the target name of your library. Note that these definitions should be \texttt{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 \texttt{HPX::plugin} target. Similarly to \texttt{HPX::component} this will set \texttt{HPX_PLUGIN_NAME} to \texttt{hpx_<target_name>}. This definition should also be \texttt{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 \texttt{hpx} subdirectory in the library install directory (typically \texttt{lib} or \texttt{lib64}). When using the \texttt{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 \texttt{*_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 \texttt{HPX_COMPONENT_PATHS} or the ini setting \texttt{hpx.component_paths} (see \texttt{--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 \texttt{add_hpx_component}. It can be used in your \texttt{CMakeLists.txt} file like this:

\begin{verbatim}
# build your application using HPX
add_hpx_component(hello_world
  SOURCES hello_world_component.cpp
  HEADERS hello_world_component.hpp
  COMPONENT_DEPENDENCIES iostreams)
\end{verbatim}
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
- **COMPIL_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:

```cpp
# 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` in your targets. The HPX include directories are available with the `HPX_INCLUDE_DIRS` CMake variable.

### 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.

---

34 https://www.cmake.org/cmake/help/latest/command/target_link_libraries.html
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:

```cpp
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\(^{35}\).

**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).

\(^{35}\) https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/RPATH-handling
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.

```cpp
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

int main()
{
    // Say hello to the world!
    hpx::cout << "Hello World!\n" << hpx::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:

```makefile
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/hpx_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)/lib/libtcmalloc_minimal.so $(HWLOC_ROOT)/lib/libhwloc.so -ldl -lrt
LINK_FLAGS=$(HPX_ROOT)/lib/hpx_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:

```bash
$ make
```

A successful build should result in hello_world binary. To test, type:
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**

```cpp
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include "hello_world_component.hpp"
#include <hpx/iostream.hpp>
#include <iostream>
namespace examples { namespace server {
  void hello_world::invoke()
  {
    hpx::cout << "Hello HPX World!" << std::endl;
  }
}}
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**

```cpp
#pragma once
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/hpx.hpp>
#include <hpx/include/actions.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/include/components.hpp>
#include <hpx/serialization.hpp>
#include <utility>
namespace examples { namespace server {
  struct HPX_COMPONENT_EXPORT hello_world
  : hpx::components::component_base<hello_world>
  {
  }
}(continues on next page)
```
void invoke();
   HPX_DEFINE_COMPONENT_ACTION(hello_world, invoke);
};
]

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::naming::id_type> && f)
   : base_type(std::move(f))
   {};

   hello_world(hpx::naming::id_type && f)
   : base_type(std::move(f))
   {};

   void invoke()
   {
      hpx::async<server::hello_world::invoke_action>(this->get_id()).get();
   }
};

#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;
}
#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.so $(HPX_ROOT)/lib/libhpx.so $(HPX_ROOT)/lib/libfilesystem.so $(BOOST_ROOT)/lib/libboost_program_options.so $(BOOST_ROOT)/lib/libboost_regex.so $(BOOST_ROOT)/lib/libboost_system.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.
2.3.3 Starting the HPX runtime

In order to write an application which 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 to block 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. It however 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:

```cpp
#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 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.

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 argv of the function main(). The list of values passed to main() will hold only the commandline options which 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 are disabled which are normally available on the HPX command line (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:
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:

```c
#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 you being able to decide which version of `hpx::init` to call. This allows to pass additional configuration parameters while initializing the `HPX` runtime system.

```c
#include <hpx/hpx_init.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 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:
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 for instance 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 to block 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 as GUIs. The function `hpx::stop` can be used to wait for the HPX runtime system to exit and should be at least 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.

```cpp
#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);

    // ...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:

```cpp
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 for instance to provide your own entry point function instead of `hpx_main`. Please refer to the function documentation for more details (see: `hpx/hpx_start.hpp`).

**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`.

```cpp
#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::apply(&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::apply([]() { 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:

```cpp
class hpx
{
public:
    static int max_idle_backoff_time = 1000;
    static int 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:

```cpp
#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::finalizes has to be called from the HPX runtime before hpx::stop
    hpx::apply([]() { 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. 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 call 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.

2.3.4 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 which is similar to the well-known Windows INI file format. This is a structured text format allowing 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 equals sign '='. The name appears to the left of the equals 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 [ and ]. 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 the '#' until the end of 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.

---

36 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>system.pid</code></td>
<td>This is initialized to store the current OS-process id of the application instance.</td>
</tr>
<tr>
<td><code>system.prefix</code></td>
<td>This is initialized to the base directory HPX has been loaded from.</td>
</tr>
<tr>
<td><code>system.executable_prefix</code></td>
<td>This is initialized to the base directory the current executable has been loaded from.</td>
</tr>
</tbody>
</table>

### 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>:
ini_path = ${[hpx.master_ini_path]}/ini
os_threads = 1
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 = ${HPX_MAX_BACKGROUND_THREADS:${hp.os_threads}}
max_idle_loop_count = ${HPX_MAX_IDLE_LOOP_COUNT:${hp_idle_loop_count_max}}
max_busy_loop_count = ${HPX_MAX_BUSY_LOOP_COUNT:${hp_busy_loop_count_max}}
max_idle_backoff_time = ${HPX_MAX_IDLE_BACKOFF_TIME:${hp_idle_backoff_time_max}}
exception_verbosity = ${HPX_EXCEPTION_VERBOSITY:2}
```

(continues on next page)
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}
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx.location</td>
<td>This is initialized to the id of the locality this application instance is running on.</td>
</tr>
<tr>
<td>hpx.component</td>
<td>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 <code>:</code> (Linux, Android, and MacOS) or using <code>;</code> (Windows).</td>
</tr>
<tr>
<td>hpx.master_ini_path</td>
<td>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 <code>:</code> (Linux, Android, and MacOS) or using <code>;</code> (Windows).</td>
</tr>
<tr>
<td>hpx.ini_path</td>
<td>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 <code>:</code> (Linux, Android, and MacOS) or using <code>;</code> (Windows).</td>
</tr>
<tr>
<td>hpx.os_threads</td>
<td>This setting reflects the number of OS-threads used for running HPX-threads. Defaults to number of detected cores (not hyperthreads/PUs).</td>
</tr>
<tr>
<td>hpx.localities</td>
<td>This setting reflects the number of localities the application is running on. Defaults to 1.</td>
</tr>
<tr>
<td>hpx.program_name</td>
<td>This setting reflects the program name of the application instance. Initialized from the command line argv[0].</td>
</tr>
<tr>
<td>hpx.command_line</td>
<td>This setting reflects the actual command line used to launch this application instance.</td>
</tr>
<tr>
<td>hpx.lock_detection</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.throw_on_exception</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.minimal_deadlock_detection_limit</td>
<td>This setting enables support for minimal deadlock detection for HPX-threads. By default this is set to 2 (for debug builds) or 0 (for Release, RelWithDebInfo, RelMinSize builds), this setting is effective only if HPX_WITH_THREAD_DEADLOCK_DETECTION is set during configuration in CMake.</td>
</tr>
<tr>
<td>hpx.spinlock_deadlock_detection_limit</td>
<td>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).</td>
</tr>
<tr>
<td>hpx.spinlock_termination</td>
<td>This setting specifies the upper limit of 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.</td>
</tr>
<tr>
<td>hpx.max_background_threads</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.max_idle_loop_count</td>
<td>By default this is defined by the preprocessor constant HPX_IDLE_LOOP_COUNT_MAX. This is an internal setting which you should change only if you know exactly what you are doing.</td>
</tr>
<tr>
<td>hpx.max_busy_loop_count</td>
<td>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 which you should change only if you know exactly what you are doing.</td>
</tr>
<tr>
<td>hpx.max_idle_backoff_time</td>
<td>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. By default this is defined by the preprocessor constant HPX_IDLE_BACKOFF_TIME_MAX. This is an internal setting which you should change only if you know exactly what you are doing.</td>
</tr>
<tr>
<td>hpx.exception_verbosity</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.small_size</td>
<td>This is initialized to the small stack size to be used by HPX-threads. Set by default to the value of the compile time preprocessor constant HPX_SMALL_STACK_SIZE (defaults to 0x20000).</td>
</tr>
<tr>
<td>hpx.medium_size</td>
<td>This is initialized to the medium stack size to be used by HPX-threads. Set by default to the value of the compile time preprocessor constant HPX_MEDIUM_STACK_SIZE (defaults to 0x20000).</td>
</tr>
<tr>
<td>hpx.large_size</td>
<td>This is initialized to the large stack size to be used by HPX-threads. Set by default to the value of the compile time preprocessor constant HPX_LARGE_STACK_SIZE (defaults to 0x20000).</td>
</tr>
<tr>
<td>hpx.huge_size</td>
<td>This is initialized to the huge stack size to be used by HPX-threads. By default this is the sum of the individual settings hpx.large_size, hpx.medium_size, and hpx.small_size.</td>
</tr>
<tr>
<td>hpx.os_threads</td>
<td>This setting reflects the number of OS-threads used for running HPX-threads. Defaults to number of detected cores (not hyperthreads/PUs).</td>
</tr>
<tr>
<td>hpx.localities</td>
<td>This setting reflects the number of localities the application is running on. Defaults to 1.</td>
</tr>
<tr>
<td>hpx.program_name</td>
<td>This setting reflects the program name of the application instance. Initialized from the command line argv[0].</td>
</tr>
<tr>
<td>hpx.command_line</td>
<td>This setting reflects the actual command line used to launch this application instance.</td>
</tr>
<tr>
<td>hpx.lock_detection</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.throw_on_exception</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.minimal_deadlock_detection_limit</td>
<td>This setting enables support for minimal deadlock detection for HPX-threads. By default this is set to 2 (for debug builds) or 0 (for Release, RelWithDebInfo, RelMinSize builds), this setting is effective only if HPX_WITH_THREAD_DEADLOCK_DETECTION is set during configuration in CMake.</td>
</tr>
<tr>
<td>hpx.spinlock_deadlock_detection_limit</td>
<td>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).</td>
</tr>
<tr>
<td>hpx.spinlock_termination</td>
<td>This setting specifies the upper limit of 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.</td>
</tr>
<tr>
<td>hpx.max_background_threads</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.max_idle_loop_count</td>
<td>By default this is defined by the preprocessor constant HPX_IDLE_LOOP_COUNT_MAX. This is an internal setting which you should change only if you know exactly what you are doing.</td>
</tr>
<tr>
<td>hpx.max_busy_loop_count</td>
<td>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 which you should change only if you know exactly what you are doing.</td>
</tr>
<tr>
<td>hpx.max_idle_backoff_time</td>
<td>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. By default this is defined by the preprocessor constant HPX_IDLE_BACKOFF_TIME_MAX. This is an internal setting which you should change only if you know exactly what you are doing.</td>
</tr>
<tr>
<td>hpx.exception_verbosity</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.small_size</td>
<td>This is initialized to the small stack size to be used by HPX-threads. Set by default to the value of the compile time preprocessor constant HPX_SMALL_STACK_SIZE (defaults to 0x20000).</td>
</tr>
<tr>
<td>hpx.medium_size</td>
<td>This is initialized to the medium stack size to be used by HPX-threads. Set by default to the value of the compile time preprocessor constant HPX_MEDIUM_STACK_SIZE (defaults to 0x20000).</td>
</tr>
<tr>
<td>hpx.large_size</td>
<td>This is initialized to the large stack size to be used by HPX-threads. Set by default to the value of the compile time preprocessor constant HPX_LARGE_STACK_SIZE (defaults to 0x20000).</td>
</tr>
<tr>
<td>hpx.huge_size</td>
<td>This is initialized to the huge stack size to be used by HPX-threads. By default this is the sum of the individual settings hpx.large_size, hpx.medium_size, and hpx.small_size.</td>
</tr>
</tbody>
</table>

---

**Chapter 2: What's so special about HPX?**
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}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx.threadpools.io_pool_size</code></td>
<td>The value of this property defines the number of OS-threads created for the internal I/O thread pool.</td>
</tr>
<tr>
<td><code>hpx.threadpools.parcel_pool_size</code></td>
<td>The value of this property defines the number of OS-threads created for the internal parcel thread pool.</td>
</tr>
<tr>
<td><code>hpx.threadpools.timer_pool_size</code></td>
<td>The value of this property defines the number of OS-threads created for the internal timer thread pool.</td>
</tr>
</tbody>
</table>

The `hpx.thread_queue` configuration section

**Important:** These settings control internal values used by the thread scheduling queues in the HPX scheduler. You should not modify these settings except if 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}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx.thread_queue.min_tasks_to_steal_pending</code></td>
<td>The value of this property defines the number of pending HPX threads which have to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.</td>
</tr>
<tr>
<td><code>hpx.thread_queue.min_tasks_to_steal_staged</code></td>
<td>The value of this property defines the number of staged HPX tasks which have to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.</td>
</tr>
<tr>
<td><code>hpx.thread_queue.min_add_new_count</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>hpx.thread_queue.max_add_new_count</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>hpx.thread_queue.max_delete_count</code></td>
<td>The value of this property defines the number of terminated HPX threads to discard during each invocation of the corresponding function.</td>
</tr>
</tbody>
</table>
The `hpx.components` configuration section

```ini
[hpx.components]
load_external = ${HPX_LOAD_EXTERNAL_COMPONENTS:1}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx.components.load_external</td>
<td>This entry defines whether external components will be loaded on this locality. This entry normally is 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.</td>
</tr>
</tbody>
</table>

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:

```ini
[hpx.components.<component_instance_name>]
name = <component_name>
path = <full_path_of_the_component_module>
enabled = ${hpx.components.load_external}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx.components.&lt;component_instance_name&gt;.name</td>
<td>This is the name of a component, usually the same as the second argument to the macro used while registering the component with <code>HPX_REGISTER_COMPONENT</code>. Set by the component factory.</td>
</tr>
<tr>
<td>hpx.components.&lt;component_instance_name&gt;.path</td>
<td>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 <code>hpx.components.&lt;component_instance_name&gt;.name</code>. Set by the component factory.</td>
</tr>
<tr>
<td>hpx.components.&lt;component_instance_name&gt;.enabled</td>
<td>This setting explicitly enables or disables the component. This is an optional property. HPX assumed that the component is enabled if it is not defined.</td>
</tr>
</tbody>
</table>

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

```ini
[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}}
```

(continues on next page)
async_serialization = ${HPX_PARCEL_ASYNC_SERIALIZATION:1}
message_handlers = ${HPX_PARCEL_MESSAGE_HANDLERS:0}

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx. parcel. address</td>
<td>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 which can be resolved into an IP address. The default depends on the compile time preprocessor constant HPX_INITIAL_IP_ADDRESS(&quot;127.0.0.1&quot;).</td>
</tr>
<tr>
<td>hpx. parcel. port</td>
<td>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).</td>
</tr>
<tr>
<td>hpx. parcel. bootstrap</td>
<td>This property defines which parcelport type should be used during application bootstrap. The default depends on the compile time preprocessor constant HPX_PARCEL_BOOTSTRAP(&quot;tcp&quot;).</td>
</tr>
<tr>
<td>hpx. parcel. max_connections</td>
<td>This property defines how many network connections between different localities are overall kept alive by each of locality. The default depends on the compile time preprocessor constant HPX_PARCEL_MAX_CONNECTIONS(512).</td>
</tr>
<tr>
<td>hpx. parcel. max_connections_per_locality</td>
<td>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).</td>
</tr>
<tr>
<td>hpx. parcel. max_message_size</td>
<td>This property defines the maximum allowed message size which will be transferable through the parcel layer. The default depends on the compile time preprocessor constant HPX_PARCEL_MAX_MESSAGE_SIZE(1000000000 bytes).</td>
</tr>
<tr>
<td>hpx. parcel. max_outbound_message_size</td>
<td>This property defines the maximum allowed outbound coalesced message size which will be transferable through the parcel layer. The default depends on the compile time preprocessor constant HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE(1000000 bytes).</td>
</tr>
<tr>
<td>hpx. parcel. array_optimization</td>
<td>This property defines whether this locality is allowed to utilize array optimizations during serialization of parcel data. The default is 1.</td>
</tr>
<tr>
<td>hpx. parcel. zero_copy_optimization</td>
<td>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.</td>
</tr>
<tr>
<td>hpx. parcel. async_serialization</td>
<td>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.</td>
</tr>
<tr>
<td>hpx. parcel. message_handlers</td>
<td>This property defines whether message handlers are loaded. The default is 0.</td>
</tr>
</tbody>
</table>

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}}
async_serialization = ${HPX_PARCEL_TCP_ASYNC_SERIALIALIZATION:${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]}

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx.parcel.tcp.enable</td>
<td>Enable the use of the default TCP parcellport. Note that the initial bootstrap of the overall HPX application will be performed using the default TCP connections. This parcellport is enabled by default. This will be disabled only if MPI is enabled (see below).</td>
</tr>
<tr>
<td>hpx.parcel.tcp.array_optimization</td>
<td>This property defines whether this locality is allowed to utilize array optimizations in the TCP/IP parcellport during serialization of parcel data. The default is the same value as set for hpx.parcel.array_optimization.</td>
</tr>
<tr>
<td>hpx.parcel.tcp.zero_copy_optimization</td>
<td>This property defines whether this locality is allowed to utilize zero copy optimizations in the TCP/IP parcellport during serialization of parcel data. The default is the same value as set for hpx.parcel.zero_copy_optimization.</td>
</tr>
<tr>
<td>hpx.parcel.tcp.async_serialization</td>
<td>This property defines whether this locality is allowed to spawn a new thread for serialization in the TCP/IP parcellport (this is both for encoding and decoding parcels). The default is the same value as set for hpx.parcel.async_serialization.</td>
</tr>
<tr>
<td>hpx.parcel.tcp.parcel_pool_size</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.parcel.tcp.max_connections</td>
<td>This property defines how many network connections between different localities are overall kept alive by each of locality. The default is taken from hpx.parcel.max_connections.</td>
</tr>
<tr>
<td>hpx.parcel.tcp.max_connections_per_locality</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.parcel.tcp.max_message_size</td>
<td>This property defines the maximum allowed message size which will be transferable through the parcel layer. The default is taken from hpx.parcel.max_message_size.</td>
</tr>
<tr>
<td>hpx.parcel.tcp.max_outbound_message_size</td>
<td>This property defines the maximum allowed outbound coalesced message size which will be transferable through the parcel layer. The default is taken from hpx.parcel.max_outbound_message_size.</td>
</tr>
</tbody>
</table>

The following settings relate to the MPI parcellport. 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}
multithreaded = ${HPX_HAVE_PARCELPORT_MPI_MULTITHREADED:0}
rank = <MPI_rank>
processor_name = <MPI_processor_name>
array_optimization = ${HPX_HAVE_PARCELPORT_MPI_ARRAY_OPTIMIZATION:$[hpx.parcel.array_optimization]}
zero_copy_optimization = ${HPX_HAVE_PARCELPORT_MPI_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.zero_copy_optimization]}
use_io_pool = ${HPX_HAVE_PARCELPORT_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]}
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx.parcel.mpi.enable</td>
<td>Enable 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.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.env</td>
<td>This property influences which environment variables (comma separated) will be analyzed to find out whether the application was invoked by MPI.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.multithreaded</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.rank</td>
<td>This property will be initialized to the MPI rank of the locality.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.processor_name</td>
<td>This property will be initialized to the MPI processor name of the locality.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.array_optimization</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.zero_copy_optimization</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.use_io_pool</td>
<td>This property can be set to run the progress thread inside of HPX threads instead of a separate thread pool. The default is 1.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.async_serialization</td>
<td>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 as set for hpx.parcel.async_serialization.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.parcel_pool_size</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.max_connections</td>
<td>This property defines how many network connections between different localities are overall kept alive by each of locality. The default is taken from hpx.parcel.max_connections.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.max_connections_per_locality</td>
<td>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.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.max_message_size</td>
<td>This property defines the maximum allowed message size which will be transferable through the parcel layer. The default is taken from hpx.parcel.max_message_size.</td>
</tr>
<tr>
<td>hpx.parcel.mpi.max_outbound_message_size</td>
<td>This property defines the maximum allowed outbound coalesced message size which will be transferable through the parcel layer. The default is taken from hpx.parcel.max_outbound_message_size.</td>
</tr>
</tbody>
</table>
The `hpx.agas` configuration section

```yaml
[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>}
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx.agas.address</td>
<td>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 <code>--hpx:agas</code> command line option). The expected format is any valid IP address or domain name format which can be resolved into an IP address. The default depends on the compile time preprocessor constant <code>HPX_INITIAL_IP_ADDRESS</code> (“127.0.0.1”).</td>
</tr>
<tr>
<td>hpx.agas.port</td>
<td>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 <code>--hpx:agas</code> command line option). The default depends on the compile time preprocessor constant <code>HPX_INITIAL_IP_PORT</code> (7009).</td>
</tr>
<tr>
<td>hpx.agas.service_mode</td>
<td>This property specifies what type of AGAS service is running on this <code>locality</code>. Currently, two modes exist. The <code>locality</code> that acts as the AGAS server runs in <code>bootstrap</code> mode. All other localities are in <code>hosted</code> mode.</td>
</tr>
<tr>
<td>hpx.agas.dedicated_server</td>
<td>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 <code>--hpx:run-agas-server-only</code> is present.</td>
</tr>
<tr>
<td>hpx.agas.max_pending_refcnt_requests</td>
<td>This property defines the number of reference counting requests (increments or decrements) to buffer. The default depends on the compile time preprocessor constant <code>HPX_AGAS_MAX_PENDING_REFCNT_REQUESTS</code> (4096).</td>
</tr>
<tr>
<td>hpx.agas.use_caching</td>
<td>This property specifies whether a software address translation cache is used. It is a boolean value. Defaults to 1.</td>
</tr>
<tr>
<td>hpx.agas.use_range_caching</td>
<td>This property specifies whether range-based caching is used by the software address translation cache. This property is ignored if <code>hpx.agas.use_caching</code> is false. It is a boolean value. Defaults to false.</td>
</tr>
<tr>
<td>hpx.agas.local_cache_size</td>
<td>This property defines the size of the software address translation cache for AGAS services. This property is ignored if <code>hpx.agas.use_caching</code> is false. Note that if <code>hpx.agas.use_range_caching</code> 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 <code>HPX_AGAS_LOCAL_CACHE_SIZE</code> (4096).</td>
</tr>
</tbody>
</table>
The `hpx.commandline` configuration section

The following table lists the definition of all pre-defined command line option shortcuts. For more information about command line 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
```
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hpx.commandline.</strong></td>
<td><strong>aliasing</strong> Enable command line aliases as defined in the section hpx.commandline. aliases (see below). Defaults to 1.</td>
</tr>
<tr>
<td></td>
<td><strong>allow_unknown</strong> Allow for unknown command line options to be passed through to hpx_main() Defaults to 0.</td>
</tr>
<tr>
<td><strong>hpx.commandline.</strong></td>
<td><strong>aliases.-a</strong> On the commandline, -a expands to: --hpx:agas.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-c</strong> On the commandline, -c expands to: --hpx:console.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-h</strong> On the commandline, -h expands to: --hpx:help.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.--help</strong> On the commandline, --help expands to: --hpx:help.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-I</strong> On the commandline, -I expands to: --hpx:ini.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-l</strong> On the commandline, -l expands to: --hpx:localities.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-p</strong> On the commandline, -p expands to: --hpx:app-config.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-q</strong> On the commandline, -q expands to: --hpx:queuing.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-r</strong> On the commandline, -r expands to: --hpx:run-agas-server.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-t</strong> On the commandline, -t expands to: --hpx:threads.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-v</strong> On the commandline, -v expands to: --hpx:version.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.--version</strong> On the commandline, --version expands to: --hpx:version.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-w</strong> On the commandline, -w expands to: --hpx:worker.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-x</strong> On the commandline, -x expands to: --hpx:hpx.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-0</strong> On the commandline, -0 expands to: --hpx:node=0.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-1</strong> On the commandline, -1 expands to: --hpx:node=1.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-2</strong> On the commandline, -2 expands to: --hpx:node=2.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-3</strong> On the commandline, -3 expands to: --hpx:node=3.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-4</strong> On the commandline, -4 expands to: --hpx:node=4.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-5</strong> On the commandline, -5 expands to: --hpx:node=5.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-6</strong> On the commandline, -6 expands to: --hpx:node=6.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-7</strong> On the commandline, -7 expands to: --hpx:node=7.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-8</strong> On the commandline, -8 expands to: --hpx:node=8.</td>
</tr>
<tr>
<td></td>
<td><strong>aliases.-9</strong> On the commandline, -9 expands to: --hpx:node=9.</td>
</tr>
</tbody>
</table>
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 using ';' (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 using ';' (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 is 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 default configuration section in the internal configuration database for each component found. For each found component the following information is generated:

```plaintext
[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 will be responsible to create new and discard 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

In this section we assume to have a simple application component which 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`:

```cpp
// 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" << hpx::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 which define boilerplate code needed for HPX to function properly:

```cpp
// 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. We create one instance of the `app::server` component on the current locality and invoke the exposed action `print_greeting_action` 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, here at the closing brace of the block surrounding the code:

```cpp
// file: $APP_ROOT/use_app_server_example.cpp
#include <hpx/hpx_init.hpp>
#include "app_server.hpp"
```

(continues on next page)
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<hpx::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, which 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.

```ini
# 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:

```ini
# 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 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 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>`, or
- 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 to log all available messages. 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 to use logging without any additional configuration effort. The following table lists the available categories.

<table>
<thead>
<tr>
<th>Category</th>
<th>Category shortcut</th>
<th>Information to be generated</th>
<th>Environment variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>None</td>
<td>Logging information generated by different subsystems of HPX, such as thread-manager, parcel layer, LCOs, etc.</td>
<td>HPX_LOGLEVEL</td>
</tr>
<tr>
<td>AGAS</td>
<td>AGAS</td>
<td>Logging output generated by the AGAS subsystem</td>
<td>HPX_AGAS_LOGLEVEL</td>
</tr>
<tr>
<td>Application</td>
<td>APP</td>
<td>Logging generated by applications.</td>
<td>HPX_APP_LOGLEVEL</td>
</tr>
</tbody>
</table>

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 environmental 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>
<thead>
<tr>
<th>Logging level</th>
<th>Integral value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;debug&gt;</td>
<td>5</td>
</tr>
<tr>
<td>&lt;info&gt;</td>
<td>4</td>
</tr>
<tr>
<td>&lt;warning&gt;</td>
<td>3</td>
</tr>
<tr>
<td>&lt;error&gt;</td>
<td>2</td>
</tr>
<tr>
<td>&lt;fatal&gt;</td>
<td>1</td>
</tr>
<tr>
<td>No logging</td>
<td>0</td>
</tr>
</tbody>
</table>

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 = ${HPX_LOGFORMAT:($%locality%/%hpxthread%/%hpxphase%/%hpxcomponent%)%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>
<thead>
<tr>
<th>Logging destination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file:&lt;filename&gt;</td>
<td>Direct all output to a file with the given &lt;filename&gt;.</td>
</tr>
<tr>
<td>cout</td>
<td>Direct all output to the local standard output of the application instance on this locality.</td>
</tr>
<tr>
<td>cerr</td>
<td>Direct all output to the local standard error output of the application instance on this locality.</td>
</tr>
<tr>
<td>console</td>
<td>Direct all output to the console instance of the application. The console instance has its logging destinations configured separately.</td>
</tr>
<tr>
<td>android_log</td>
<td>Direct all output to the (Android) system log (available on Android systems only).</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locality</td>
<td>The id of the locality on which the logging message was generated.</td>
</tr>
<tr>
<td>hpxthread</td>
<td>The id of the HPX-thread generating this logging output.</td>
</tr>
<tr>
<td>hpxphase</td>
<td>The phase of the HPX-thread generating this logging output.</td>
</tr>
<tr>
<td>hpxcomponent</td>
<td>The local virtual address of the component which the current HPX-thread is accessing.</td>
</tr>
<tr>
<td>parentloc</td>
<td>The id of the locality where the HPX thread was running which initiated the current HPX-thread.</td>
</tr>
<tr>
<td>hpxparent</td>
<td>The id of the HPX-thread which initiated the current HPX-thread. The current HPX-thread is generating this logging output.</td>
</tr>
<tr>
<td>hpxparentphase</td>
<td>The phase of the HPX-thread when it initiated the current HPX-thread. The current HPX-thread is generating this logging output.</td>
</tr>
<tr>
<td>time</td>
<td>The time stamp for this logging output line as generated by the source locality.</td>
</tr>
<tr>
<td>idx</td>
<td>The sequence number of the logging output line as generated on the source locality.</td>
</tr>
<tr>
<td>osthread</td>
<td>The sequence number of the OS-thread which executes the current HPX-thread.</td>
</tr>
</tbody>
</table>

**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 line break):

```
(T00000000/0000000002d46f90.01/0000000009ebc10) P--------/0000000002d46f80.02 17:49.320 [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 = ${HP_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 which name is generated based from 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.

---

38 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:hp** 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: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
the 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 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 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
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

<table>
<thead>
<tr>
<th>Shortcut option</th>
<th>Equivalent long option</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>--hpx:agas</td>
</tr>
<tr>
<td>-c</td>
<td>--hpx:console</td>
</tr>
<tr>
<td>-h</td>
<td>--hpx:help</td>
</tr>
<tr>
<td>-I</td>
<td>--hpx:ini</td>
</tr>
<tr>
<td>-l</td>
<td>--hpx:localities</td>
</tr>
<tr>
<td>-p</td>
<td>--hpx:app-config</td>
</tr>
<tr>
<td>-q</td>
<td>--hpx:queuing</td>
</tr>
<tr>
<td>-r</td>
<td>--hpx:run-agas-server</td>
</tr>
<tr>
<td>-t</td>
<td>--hpx:threads</td>
</tr>
<tr>
<td>-v</td>
<td>--hpx:version</td>
</tr>
<tr>
<td>-w</td>
<td>--hpx:worker</td>
</tr>
<tr>
<td>-x</td>
<td>--hpx:hpx</td>
</tr>
<tr>
<td>-0</td>
<td>--hpx:node=0</td>
</tr>
<tr>
<td>-1</td>
<td>--hpx:node=1</td>
</tr>
<tr>
<td>-2</td>
<td>--hpx:node=2</td>
</tr>
<tr>
<td>-3</td>
<td>--hpx:node=3</td>
</tr>
<tr>
<td>-4</td>
<td>--hpx:node=4</td>
</tr>
<tr>
<td>-5</td>
<td>--hpx:node=5</td>
</tr>
<tr>
<td>-6</td>
<td>--hpx:node=6</td>
</tr>
<tr>
<td>-7</td>
<td>--hpx:node=7</td>
</tr>
<tr>
<td>-8</td>
<td>--hpx:node=8</td>
</tr>
<tr>
<td>-9</td>
<td>--hpx:node=9</td>
</tr>
</tbody>
</table>

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 which 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 options 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 (Portable Hardware Locality (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:

```plaintext
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 for which to define affinity bindings, and core:0-3.pu:0: 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:
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 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.

2.3.5 Writing single-node HPX applications

HPX is a C++ Standard Library for Concurrency and Parallelism. This means that it implements all of the corresponding facilities as defined by the C++ Standard. Additionally, HPX implements functionalities 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.

Using LCOs

Lightweight Control Objects (LCOs) provide synchronization for HPX applications. Most of them are familiar from other frameworks, but a few of them work in slightly different ways adapted to HPX. The following synchronization objects are available in HPX:

1. future
2. queue
3. object_semaphore
4. barrier

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:

```cpp
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:

```cpp
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();
}
```

(continues on next page)

---

37 https://en.wikipedia.org/wiki/Message_Passing_Interface
void send_receive_channel()
{
    hpx::lcos::local::channel<int> c;
    hpx::lcos::local::channel<> done;

    hpx::apply(&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:

```cpp
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::apply(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();
}
```
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):

```cpp
hpx::lcos::local::guard gu;
run_guarded(gu, task);
```

If a single method needs to run with multiple guards, use a guard set:

```cpp
boost::shared<hpx::lcos::local::guard> gu1(new hpx::lcos::local::guard());
boost::shared<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. The following guards are available in HPX:

1. conditional_trigger
2. counting_semaphore
3. dataflow
4. event
5. mutex
6. once
7. recursive_mutex
8. spinlock
9. spinlock_no_backoff
10. trigger

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 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:

```cpp
#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 (as proposed by __cpp11_n4107__). 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

<table>
<thead>
<tr>
<th>Facility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx::future::then</td>
<td>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.</td>
</tr>
<tr>
<td>unwrap constructor for hpx::future</td>
<td>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.</td>
</tr>
<tr>
<td>hpx::future::is_ready</td>
<td>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.</td>
</tr>
<tr>
<td>hpx::make_ready_future</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

In addition to the extensions proposed by N4313\(^\text{39}\), HPX adds functions allowing users to compose several futures in a more flexible way.

\(^{39}\) [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html)
Table 2.12: Facilities for composing `hpx::future`

<table>
<thead>
<tr>
<th>Facility</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx::when_any</code></td>
<td>Asynchronously wait for at least one of multiple future or shared_future objects to finish.</td>
<td>N431340, ..._n versions are HPX only</td>
</tr>
<tr>
<td><code>hpx::wait_any</code></td>
<td>Synchronously wait for at least one of multiple future or shared_future objects to finish.</td>
<td>HPX only</td>
</tr>
<tr>
<td><code>hpx::when_all</code></td>
<td>Asynchronously wait for all future and shared_future objects to finish.</td>
<td>N431341, ..._n versions are HPX only</td>
</tr>
<tr>
<td><code>hpx::wait_all</code></td>
<td>Synchronously wait for all future and shared_future objects to finish.</td>
<td>HPX only</td>
</tr>
<tr>
<td><code>hpx::when_some</code></td>
<td>Asynchronously wait for multiple future and shared_future objects to finish.</td>
<td>HPX only</td>
</tr>
<tr>
<td><code>hpx::wait_some</code></td>
<td>Synchronously wait for multiple future and shared_future objects to finish.</td>
<td>HPX only</td>
</tr>
<tr>
<td><code>hpx::when_each</code></td>
<td>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.</td>
<td>HPX only</td>
</tr>
<tr>
<td><code>hpx::wait_each</code></td>
<td>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.</td>
<td>HPX only</td>
</tr>
</tbody>
</table>

**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 N440942 (Working Draft, Technical Specification for C++ Extensions for Parallelism), N441143 (Task Blocks), and N440644 (Parallel Algorithms Need Executors).

**Using parallel algorithms**

A parallel algorithm is a function template described by this document which is declared in the (inline) namespace `hpx::parallel::v1`.

Note: For compilers that do not support inline namespaces, all of the namespace v1 is imported into the namespace `hpx::parallel`. The effect is similar to what inline namespaces would do, namely all names defined in `hpx::parallel::v1` are accessible from the namespace `hpx::parallel` as well.

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

---

40 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
41 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
42 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf
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 applications of function objects in parallel algorithms invoked with an execution policy of type `hpx::execution::parallel_unsequenced_policy` is, in HPX, equivalent to the use of the execution policy `hpx::execution::parallel_policy`.

Algorithms invoked with an execution policy object of type `hpx::parallel::v1::execution_policy` execute internally as if invoked with the contained execution policy object. No exception is thrown when an `hpx::parallel::v1::execution_policy` contains an execution policy of type `hpx::execution::sequenced_task_policy` or `hpx::execution::parallel_task_policy` (which normally turn the algorithm into its asynchronous version). In this case the execution is semantically equivalent to the case of passing a `hpx::execution::sequenced_policy` or `hpx::execution::parallel_policy` contained in the `hpx::parallel::v1::execution_policy` object respectively.

### 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` object.

For example, the number of invocations of the user-provided function object in `for_each` is unspecified. When `hpx::parallel::v1::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 (in header: `<hpx/algorithm.hpp>`)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
<th>Algorithm page at cppreference.com</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx::adjacent_find</code></td>
<td>Computes the differences between adjacent elements in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>adjacent_find</code></td>
</tr>
<tr>
<td><code>hpx::all_of</code></td>
<td>Checks if a predicate is <code>true</code> for all of the elements in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>all_any_none_of</code></td>
</tr>
<tr>
<td><code>hpx::any_of</code></td>
<td>Checks if a predicate is <code>true</code> for any of the elements in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>all_any_none_of</code></td>
</tr>
<tr>
<td><code>hpx::count</code></td>
<td>Returns the number of elements equal to a given value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>count</code></td>
</tr>
<tr>
<td><code>hpx::count_if</code></td>
<td>Returns the number of elements satisfying a specific criteria.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>count_if</code></td>
</tr>
<tr>
<td><code>hpx::equal</code></td>
<td>Determines if two sets of elements are the same.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>equal</code></td>
</tr>
<tr>
<td><code>hpx::find</code></td>
<td>Finds the first element equal to a given value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>find</code></td>
</tr>
<tr>
<td><code>hpx::find_end</code></td>
<td>Finds the last sequence of elements in a certain range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>find_end</code></td>
</tr>
<tr>
<td><code>hpx::find_first_of</code></td>
<td>Searches for any one of a set of elements.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>find_first_of</code></td>
</tr>
<tr>
<td><code>hpx::find_if</code></td>
<td>Finds the first element satisfying a specific criteria.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>find_if</code></td>
</tr>
<tr>
<td><code>hpx::find_if_not</code></td>
<td>Finds the first element not satisfying a specific criteria.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>find_if_not</code></td>
</tr>
<tr>
<td><code>hpx::for_each</code></td>
<td>Applies a function to a range of elements.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>for_each</code></td>
</tr>
<tr>
<td><code>hpx::for_each_n</code></td>
<td>Applies a function to a number of elements.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>for_each_n</code></td>
</tr>
<tr>
<td><code>hpx::lexicographical_compare</code></td>
<td>Checks if a range of values is lexicographically less than another range of values.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>lexicographical_compare</code></td>
</tr>
<tr>
<td><code>hpx::parallel::v1::mismatch</code></td>
<td>Finds the first position where two ranges differ.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>mismatch</code></td>
</tr>
<tr>
<td><code>hpx::none_of</code></td>
<td>Checks if a predicate is <code>true</code> for none of the elements in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>all_any_none_of</code></td>
</tr>
<tr>
<td><code>hpx::search</code></td>
<td>Searches for a range of elements.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>search</code></td>
</tr>
<tr>
<td><code>hpx::search_n</code></td>
<td>Searches for a number consecutive copies of an element in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>search_n</code></td>
</tr>
</tbody>
</table>
Chapter 2. What’s so special about HPX?
## Table 2.14: Modifying parallel algorithms (In Header: `<hpx/algorithm.hpp>`)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
<th>Algorithm page at cppreference.com</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx::copy</code></td>
<td>Copies a range of elements to a new location.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>exclusive_scan</code>63</td>
</tr>
<tr>
<td><code>hpx::copy_n</code></td>
<td>Copies a number of elements to a new location.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>copy_n</code>64</td>
</tr>
<tr>
<td><code>hpx::copy_if</code></td>
<td>Copies the elements from a range to a new location for which the given predicate is <code>true</code></td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>copy</code>85</td>
</tr>
<tr>
<td><code>hpx::move</code></td>
<td>Moves a range of elements to a new location.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>move</code>86</td>
</tr>
<tr>
<td><code>hpx::fill</code></td>
<td>Assigns a range of elements a certain value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>fill</code>97</td>
</tr>
<tr>
<td><code>hpx::fill_n</code></td>
<td>Assigns a value to a number of elements.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>fill_n</code>80</td>
</tr>
<tr>
<td><code>hpx::generate</code></td>
<td>Saves the result of a function in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>generate</code>99</td>
</tr>
<tr>
<td><code>hpx::generate_n</code></td>
<td>Saves the result of N applications of a function.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>generate_n</code>99</td>
</tr>
<tr>
<td><code>hpx::remove</code></td>
<td>Removes the elements from a range that are equal to the given value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>remove</code>71</td>
</tr>
<tr>
<td><code>hpx::remove_if</code></td>
<td>Removes the elements from a range that are equal to the given predicate is <code>false</code></td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>remove</code>72</td>
</tr>
<tr>
<td><code>hpx::remove_copy</code></td>
<td>Copies the elements from a range to a new location that are not equal to the given value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>remove_copy</code>74</td>
</tr>
<tr>
<td><code>hpx::replace</code></td>
<td>Replaces all values satisfying specific criteria with another value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>replace</code>75</td>
</tr>
<tr>
<td><code>hpx::replace_if</code></td>
<td>Replaces all values satisfying specific criteria with another value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>replace</code>76</td>
</tr>
<tr>
<td><code>hpx::replace_copy</code></td>
<td>Copies a range, replacing elements satisfying specific criteria with another value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>replace_copy</code>77</td>
</tr>
<tr>
<td><code>hpx::replace_copy_if</code></td>
<td>Copies a range, replacing elements satisfying specific criteria with another value.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>replace_copy</code>78</td>
</tr>
<tr>
<td><code>hpx::reverse</code></td>
<td>Reverses the order elements in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>reverse</code>99</td>
</tr>
<tr>
<td><code>hpx::reverse_copy</code></td>
<td>Creates a copy of a range that is reversed.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>reverse_copy</code>80</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>In header</td>
<td>Algorithm page at cppreference.com</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
<td>----------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>hpx::merge</td>
<td>Merges two sorted ranges.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>merge89</td>
</tr>
<tr>
<td>hpx::inplace_merge</td>
<td>Merges two ordered ranges in-place.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>inplace_merge90</td>
</tr>
<tr>
<td>hpx::includes</td>
<td>Returns true if one set is a subset of another.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>includes91</td>
</tr>
<tr>
<td>hpx::set_difference</td>
<td>Computes the difference between two sets.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>set_difference92</td>
</tr>
<tr>
<td>hpx::set_intersection</td>
<td>Computes the intersection of two sets.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>set_intersection93</td>
</tr>
<tr>
<td>hpx::set_symmetric_difference</td>
<td>Computes the symmetric difference between two sets.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>set_symmetric_difference94</td>
</tr>
<tr>
<td>hpx::set_union</td>
<td>Computes the union of two sets.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>set_union95</td>
</tr>
</tbody>
</table>

84 http://en.cppreference.com/w/cpp/algorithm/copy nth
89 http://en.cppreference.com/w/cpp/algorithm/generate
91 http://en.cppreference.com/w/cpp/algorithm/remove
100 http://en.cppreference.com/w/cpp/algorithm/replacement nth
108 Chapter 2. What’s so special about HPX?
### Table 2.16: Heap operations (In Header: `<hpx/algorithm.hpp>`)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
<th>Algorithm page at cppreference.com</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx::is_heap</code></td>
<td>Returns <code>true</code> if the range is max heap.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>is_heap</code></td>
</tr>
<tr>
<td><code>hpx::is_heap_until</code></td>
<td>Returns the first element that breaks a max heap.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>is_heap_until</code></td>
</tr>
<tr>
<td><code>hpx::make_heap</code></td>
<td>Constructs a max heap in the range <code>[first, last)</code>.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>make_heap</code></td>
</tr>
</tbody>
</table>

### Table 2.17: Minimum/maximum operations (In Header: `<hpx/algorithm.hpp>`)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
<th>Algorithm page at cppreference.com</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx::parallel::v1::max_element</code></td>
<td>Returns the largest element in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>max_element</code></td>
</tr>
<tr>
<td><code>hpx::parallel::v1::min_element</code></td>
<td>Returns the smallest element in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>min_element</code></td>
</tr>
<tr>
<td><code>hpx::parallel::v1::minmax_element</code></td>
<td>Returns the smallest and the largest element in a range.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
<td><code>minmax_element</code></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
<th>Algorithm page at cppreference.com</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx::adjacent_difference</td>
<td>Calculates the difference between each element in an input range and the preceding element.</td>
<td>&lt;hpx/numeric.hpp&gt;</td>
<td>adjacent_difference[^102]</td>
</tr>
<tr>
<td>hpx::exclusive_scan</td>
<td>Does an exclusive parallel scan over a range of elements.</td>
<td>&lt;hpx/numeric.hpp&gt;</td>
<td>exclusive_scan[^103]</td>
</tr>
<tr>
<td>hpx::reduce</td>
<td>Sums up a range of elements.</td>
<td>&lt;hpx/numeric.hpp&gt;</td>
<td>reduce[^104]</td>
</tr>
<tr>
<td>hpx::inclusive_scan</td>
<td>Does an inclusive parallel scan over a range of elements.</td>
<td>&lt;hpx/algorithm.hpp&gt;</td>
<td>inclusive_scan[^105]</td>
</tr>
<tr>
<td>hpx::parallel::v1::reduce_by_key</td>
<td>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 (1,1,1,2,3,3,3,1) 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}.</td>
<td>&lt;hpx/numeric.hpp&gt;</td>
<td></td>
</tr>
<tr>
<td>hpx::transform_reduce</td>
<td>Sums up a range of elements after applying a function. Also, accumulates the inner products of two input ranges.</td>
<td>&lt;hpx/numeric.hpp&gt;</td>
<td>transform_reduce[^106]</td>
</tr>
<tr>
<td>hpx::transform_inclusive_scan</td>
<td>Does an inclusive parallel scan over a range of elements after applying a function.</td>
<td>&lt;hpx/numeric.hpp&gt;</td>
<td>transform_inclusive_scan[^107]</td>
</tr>
<tr>
<td>hpx::transform exclusive_scan</td>
<td>Does an exclusive parallel scan over a range of elements after applying a function.</td>
<td>&lt;hpx/numeric.hpp&gt;</td>
<td>transform exclusive_scan[^108]</td>
</tr>
</tbody>
</table>

Table 2.19: Dynamic Memory Management (In Header: `<hpx/memory.hpp>`)  

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
<th>Algorithm page at cppreference.com</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx::destroy</code></td>
<td>Destroys a range of objects.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>destroy[^109]</td>
</tr>
<tr>
<td><code>hpx::destroy_n</code></td>
<td>Destroys a range of objects.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>destroy_n[^110]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_copy</code></td>
<td>Copies a range of objects to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_copy[^111]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_copy_n</code></td>
<td>Copies a number of objects to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_copy_n[^112]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_default_construct</code></td>
<td>Copies a range of objects to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_default_construct[^113]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_default_construct_n</code></td>
<td>Copies a number of objects to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_default_construct_n[^114]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_fill</code></td>
<td>Copies an object to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_fill[^115]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_fill_n</code></td>
<td>Copies an object to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_fill_n[^116]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_move</code></td>
<td>Moves a range of objects to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_move[^117]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_move_n</code></td>
<td>Moves a number of objects to an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_move_n[^118]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_value_construct</code></td>
<td>Constructs objects in an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_value_construct[^119]</td>
</tr>
<tr>
<td><code>hpx::uninitialized_value_construct_n</code></td>
<td>Constructs objects in an uninitialized area of memory.</td>
<td><code>&lt;hpx/memory.hpp&gt;</code></td>
<td>uninitialized_value_construct_n[^120]</td>
</tr>
</tbody>
</table>

Table 2.20: Index-based for-loops (In Header: `<hpx/algorithm.hpp>`)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hpx::for_loop</code></td>
<td>Implements loop functionality over a range specified by integral or iterator bounds.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
</tr>
<tr>
<td><code>hpx::for_loop_strided</code></td>
<td>Implements loop functionality over a range specified by integral or iterator bounds.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
</tr>
<tr>
<td><code>hpx::for_loop_n</code></td>
<td>Implements loop functionality over a range specified by integral or iterator bounds.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
</tr>
<tr>
<td><code>hpx::for_loop_n_strided</code></td>
<td>Implements loop functionality over a range specified by integral or iterator bounds.</td>
<td><code>&lt;hpx/algorithm.hpp&gt;</code></td>
</tr>
</tbody>
</table>

**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 `hpx::parallel::executor_parameter_traits` type.

With `executor_parameter_traits`, clients access all types of executor parameters uniformly:

```cpp
std::size_t chunk_size =
    executorParameterTraits<my_parameter_t>::get_chunk_size(my_parameter,
        my_executor, []() { return 0; }, 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 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::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::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::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::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::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::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::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.

### Using task blocks

The `define_task_block`, `run` and the `wait` functions implemented based on N4411\(^{121}\) are based on the `task_block` concept that is a part of the common subset of the Microsoft Parallel Patterns Library (PPL)\(^{122}\) and the Intel Threading Building Blocks (TBB)\(^{123}\) 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++\(^{124}\) and async and finish from X10\(^{125}\). 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:

```cpp
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::parallel::define_task_block` and the `hpx::parallel::task_block::run` member function of `hpx::parallel::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

---

\(^{121}\) [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf)


\(^{123}\) [https://www.threadingbuildingblocks.org/](https://www.threadingbuildingblocks.org/)


\(^{125}\) [https://x10-lang.org/](https://x10-lang.org/)
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`:

```cpp
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 N4411\textsuperscript{126}. 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:

```cpp
template <typename Func>
int traverse(node *n, Func&& compute)
{
    int left = 0, right = 0;

    define_task_block(
        execution::par, // execution::parallel_policy
        [s](task_block<> & tb) {
            if (n->left)
                tb.run([s] { left = traverse(n->left, compute); });
            if (n->right)
                tb.run([s] { right = traverse(n->right, compute); });
        });
    return left + right;
}
```

\textsuperscript{126} http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
This also causes the `hpx::parallel::v2::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`.

HPX still supports calling `hpx::parallel::v2::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:

```cpp
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::parallel::v2::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::parallel::v2::define_task_block`.

2.3. Manual 115
2.3.6 Writing distributed HPX 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 is exposing 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: retrieve the global address of the locality this function is called on.
- hpx::find_all_localities: retrieve the global addresses of all localities available to this application (including the locality the function is being called on).
- hpx::find_remote_localities: retrieve 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: retrieve the number of localities available to this application.
- hpx::find_locality: retrieve the global address of any locality supporting the given component type.
- hpx::get_colocation_id: retrieve 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_: Create 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.
Applying actions

Action type definition

Actions are special types we use 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 \texttt{HPX\_PLAIN\_ACTION} can be used to define the action type. Here is an example demonstrating this:

```cpp
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);
```

\textbf{Important:} The macro \texttt{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 (\texttt{HPX\_DEFINE\_PLAIN\_ACTION} and \texttt{HPX\_REGISTER\_ACTION}) as shown in the next example:

```cpp
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 \texttt{some_global_action} inside the namespace \texttt{app}.

\textbf{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 \texttt{some\_global\_action}).
Important: The second argument passed to \texttt{HPX\_REGISTER\_ACTION} (\texttt{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 \texttt{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:

```cpp
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 into a source file (e.g. the main application source file) in the simplest case:

```cpp
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 
// \texttt{HPX\_REGISTER\_ACTION\_DECLARATION()} macro invocation above
typedef some_component::some_member_action some_component_some_action;
HPX_REGISTER_ACTION(some_component_some_action);

Granted, these macro invocations are a bit more complex than for simple global functions, however we believe they are still manageable.

The most important macro invocation is the \texttt{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 [hpx\_link examples/quickstart/component_in_executable.cpp..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 ‘applying the action’. Actions can have arguments, which will be supplied while the action is applied. At the minimum, one parameter is required to apply 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 apply 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. ?? 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.

This figure shows that HPX allows the user to apply actions with a syntax similar to the C++ standard. In fact, all action types have an overloaded function operator allowing to synchronously apply 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 applying 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.

<table>
<thead>
<tr>
<th>R f(p...)</th>
<th>Synchronous Execution (returns R)</th>
<th>Asynchronous Execution (returns future&lt;R&gt;)</th>
<th>Fire &amp; Forget Execution (returns void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions (direct invocation)</td>
<td>f(p...)</td>
<td>async(f, p...)</td>
<td>apply(f, p...)</td>
</tr>
<tr>
<td>Functions (lazy invocation)</td>
<td>bind(f, p...)(...)</td>
<td>async(bind(f, p...), ...)</td>
<td>apply(bind(f, p...), ...)</td>
</tr>
<tr>
<td>Actions (direct invocation)</td>
<td>HEX_ACTION(f, action) a(id, p...)</td>
<td>HEX_ACTION(f, action) async(a, id, p...)</td>
<td>HEX_ACTION(f, action) apply(a, id, p...)</td>
</tr>
<tr>
<td>Actions (lazy invocation)</td>
<td>HEX_ACTION(f, action) bind(a, id, p...) (...)</td>
<td>HEX_ACTION(f, action) async(bind(a, id, p...), ...)</td>
<td>HEX_ACTION(f, action) apply(bind(a, id, p...), ...)</td>
</tr>
</tbody>
</table>

Fig. 2.8: Overview of the main API exposed by HPX.
Additionally, HPX exposes `hpx::apply` 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.

Applying 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. Applying the action does not wait for the function to start running, instead it is a fully asynchronous operation. The following example shows how to apply the action as defined in the previous section on the local locality (the locality this code runs on):

```cpp
some_global_action act; // define an instance of some_global_action
hpx::apply(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:

```cpp
some_component_action act; // define an instance of some_component_action
hpx::apply(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.

Applying an action asynchronously with synchronization

This method will make sure the action is scheduled to run on the target locality. Applying 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::apply` 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 apply the action from above on the local locality:

```cpp
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:

```cpp
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.

### Applying 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:

```cpp
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:

```cpp
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 applying 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 applying 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.
Applying an action with a continuation but without any synchronization

This method is very similar to the method described in section Applying 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::apply` invokes a single function using ‘fire and forget’ semantics, `hpx::apply_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.

Applying an action with a continuation and with synchronization

This method is very similar to the method described in section Applying 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:

```cpp
// first action
default(int32_t i)
{
        return i+1;
}
HPX.PLAIN_ACTION(default);

// second action
default(int32_t i)
{
        return i*2;
}
HPX.PLAIN_ACTION(default);

// this code invokes 'default' above and passes along a continuation // function which will forward the result returned from 'default' to // 'default'.
default1_type act1; // define an instance of 'default1_type'
default2_type act2; // define an instance of 'default2_type'
hpx::future<int> f =
        hpx::async_continue(act1, hpx::make_continuation(act2),
               hpx::find_here(), i);
hpx::cout << f.get() << "; // 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:

```cpp
// this code invokes 'default' above and passes along a continuation // function which will forward the result returned from 'default' to // 'default'.
default1_type act1; // define an instance of 'default1_type'
default2_type act2; // define an instance of 'default2_type'
hpx::future<int> f =
        hpx::async_continue(act1, hpx::make_continuation(act2, hpx::find_here()),
```

(continues on next page)
Similarly, it is possible to chain more than 2 operations:

```cpp
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);
```

The function `hpx::make_continuation` creates a special function object which exposes the following prototype:

```cpp
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:

```cpp
namespace app
{
    void some_function_with_error(int arg)
    {
        if (arg < 0) {
            HPX_THROW_EXCEPTION(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::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:

```cpp
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()`:

```cpp
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:

```cpp
// 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);
    }
}
```

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:

```cpp
// source file some_component.cpp
#include "some_component.hpp"

// The following code generates all necessary boiler plate to enable the
```
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:

```cpp
#include <hpx/include/components.hpp>

namespace app {
    // Define a client side representation type for the component type 'some_component' defined in the previous section.
    // 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);
}
```

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:

```cpp
// 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), ...);```

```cpp
// 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`:

```cpp
// 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), ...);

// 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

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:

```cpp
#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:

```cpp
#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

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 );
```

(continues on next page)
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:

```cpp
#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>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In header</th>
<th>Class page at cppreference.com</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx::partitioned_vector</td>
<td>Dynamic segmented contiguous array.</td>
<td>&lt;hpx/include/partitioned_vector.hpp&gt;</td>
<td><code>vector&lt;...&gt;</code></td>
</tr>
</tbody>
</table>
### 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 dereferencing 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):

```cpp
#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;

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)
    {
    // some code
    }
}
```


Which is equivalent to:

```cpp
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:

```cpp
#include <hpx/collectives/spmd_block.hpp>

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */) {
    // Parallel section
    /* Do some code */
}
```

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:

- [def Team information] `get_num_images`, `this_image`, `images_per_locality`
- [def Control statements] `sync_all`, `sync_images`

Here is a sample code summarizing the features offered by the `spmd_block` class:

```cpp
#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);
    // 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 \[v[i, j]\] which would call internally the expression \[v[k]\].

By default, the partitioned_vector class integrates 1-D views of its segments:

```cpp
#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:

```cpp
#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>;

/* 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:

```cpp
#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);
```

(continues on next page)
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();
    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:

```cpp
#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
    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(136 Chapter 2. What’s so special about HPX?
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;
}

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:

```cpp
#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 2.23: Parameters of coarray constructor

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>block</code></td>
<td>Reference to a spmd_block object</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Symbolic name of type std::string</td>
</tr>
<tr>
<td><code>{height, width, _}</code></td>
<td>Dimensions of the coarray object</td>
</tr>
<tr>
<td><code>segment_size</code></td>
<td>Size of a co-indexed element (i.e. size of the object referenced by the expression <code>a(i, j, k)</code>)</td>
</tr>
</tbody>
</table>

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:

```cpp
#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++)
        for (std::size_t i = 0; i<height; i++)
            (continues on next page)```
{  
  // 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().

### 2.3.7 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 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.

```bash
#!/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:

```bash
$ pbsdsh -u $APP_PATH --hpx:nodes`cat $PBS_NODEFILE` --hpx:endnodes $APP_OPTIONS
```

129 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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/bin/bash
#
#PBS -l nodes=2:ppn=4
APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

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:
If the job is accepted, qsub will print out the assigned job ID, which may look like:

```bash
$ 42.supercomputer.some.university.edu
```

To check the status of your job, issue the following command:

```bash
$ 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:

<table>
<thead>
<tr>
<th>Job id</th>
<th>Name</th>
<th>User</th>
<th>Time</th>
<th>Use</th>
<th>S</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>42.supercomputer...ello_world.sh</td>
<td>joe_user</td>
<td>0</td>
<td>Q</td>
<td></td>
<td></td>
<td>batch</td>
</tr>
</tbody>
</table>

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
different outputs...
```

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:

```bash
$ 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 as if you were in an interactive shell.

2.3.8 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 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:

```
$ 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. Manual

143
2.3.9 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 fullinstancename distinguishes different counter instances of the same counter type. The formatting of the fullinstancename 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>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>This wildcard character matches any number (zero or more) of arbitrary characters.</td>
</tr>
<tr>
<td>?</td>
<td>This wildcard character matches any single arbitrary character.</td>
</tr>
<tr>
<td>[...]</td>
<td>This wildcard character matches any single character from the list of specified within the square brackets.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--hpx:print-counter</td>
<td>Prints the specified performance counter either repeatedly and/or at the times specified by <code>--hpx:print-counter-at</code> (see also option <code>--hpx:print-counter-interval</code>).</td>
</tr>
<tr>
<td>--hpx:print-counter-reset</td>
<td>Prints the specified performance counter either repeatedly and/or at the times specified by <code>--hpx:print-counter-at</code>. Reset the counter after the value is queried (see also option <code>--hpx:print-counter-interval</code>).</td>
</tr>
<tr>
<td>--hpx:print-counter-interval</td>
<td>Prints the performance counter(s) specified with <code>--hpx:print-counter</code> repeatedly after the time interval (specified in milliseconds) (default: 0 which means print once at shutdown).</td>
</tr>
<tr>
<td>--hpx:print-counter-destination</td>
<td>Prints the performance counter(s) specified with <code>--hpx:print-counter</code> to the given file (default: console).</td>
</tr>
<tr>
<td>--hpx:list-counters</td>
<td>Lists the names of all registered performance counters.</td>
</tr>
<tr>
<td>--hpx:list-counter-infos</td>
<td>Lists the description of all registered performance counters.</td>
</tr>
<tr>
<td>--hpx:print-counter-format</td>
<td>Prints the performance counter(s) specified with <code>--hpx:print-counter</code>. Possible formats in CVS format with header or without any header (see option <code>--hpx:no-csv-header</code>), 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 <code>--hpx:print-counter-shortname,full-countername</code>).</td>
</tr>
<tr>
<td>--hpx:no-csv-header</td>
<td>Prints the performance counter(s) specified with <code>--hpx:print-counter</code> and csv or csv-short format specified with <code>--hpx:print-counter-format</code> without header.</td>
</tr>
<tr>
<td>--hpx:print-counter-at arg</td>
<td>Prints the performance counter(s) specified with <code>--hpx:print-counter</code> (or <code>--hpx:print-counter-reset</code>) at the given point in time. Possible argument values: startup, shutdown (default), noshutdown.</td>
</tr>
<tr>
<td>--hpx:reset-counters</td>
<td>Resets all performance counter(s) specified with <code>--hpx:print-counter</code> after they have been evaluated.</td>
</tr>
<tr>
<td>--hpx:print-counter-types</td>
<td>Appends counter type description to generated output.</td>
</tr>
<tr>
<td>--hpx:print-counters-locally</td>
<td>Each locality prints only its own local counters.</td>
</tr>
</tbody>
</table>

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#*/counter/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_raw version: 1.0.0
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
fullname: /agas/count/bind helptext: returns the number of invocations of the AGAS service 'bind' type: counter_raw version: 1.0.0
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
fullname: /agas/count/bind_gid helptext: returns the number of invocations of the AGAS service 'bind_gid' type: counter_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:

```bash
$ 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

```bash
$ 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,1,0.002543,[s],33
/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:

```bash
$ 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:

```bash
$ 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`:

```bash
$ 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:

```bash
$ 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:
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:

```cpp
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:

```cpp
// 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() << hpx::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:

```cpp
// print the current number of threads created on locality 0
hpx::performance_counters::performance_counter count("/threads/locality#0/total/count/cumulative");
hpx::future<int> result = count.get_value<int>();
hpx::cout << result.get() << hpx::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:

```cpp
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:

```cpp
// 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:

```cpp
#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
    );
}
```

(continues on next page)
"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:

```cpp
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:

```cpp
// Copyright (c) 2007-2020 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>

namespace hpx { namespace 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;

        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(std::move(id))
        {
        }

        performance_counter(hpx::future<performance_counter>&& c)
            : base_type(std::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 = false);
        counter_value get_counter_value(

        (continues on next page)
launch::sync_policy, bool reset = false, error_code& ec = throws);

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 = false);
counter_values_array get_counter_values_array(
  launch::sync_policy, bool reset = false, error_code& ec = throws);

global_value get_global_value() const;
global_value get_global_value(
  launch::sync_policy, bool reset = false, 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();
bool start(launch::sync_policy, error_code& ec = throws);

future<bool> stop();
bool stop(launch::sync_policy, error_code& ec = throws);

future<void> reset();
void reset(launch::sync_policy, error_code& ec = throws);

future<void> reinit(bool reset = true);
void reinit(
  launch::sync_policy, bool reset = true, error_code& ec = throws);

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>();
  }

public:

  template <typename T>
  future<T> get_value(bool reset = false)
  {
    return get_counter_value(reset).then(hpx::launch::sync,
      util::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

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 `hpx/performance_counters/base_performance_counter.hpp` as:

```
namespace hpx::performance_counters {
    template<typename Derived>
    class base_performance_counter
    {
        private:
            typedef hpx::components::component_base<Derived> base_type;

        template <typename T>
        T get_value(launch::sync_policy, error_code& ec = throws) const
        {
            return get_counter_value(launch::sync).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);
    }
}
```

(continues on next page)
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
//
#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 
{ 

  class sine_counter : public hpx::performance_counters::base_performance_counter<sine_counter>
  { 

    public:
    typedef Derived type_holder;
    typedef hpx::performance_counters::server::base_performance_counter base_type_holder;

    base_performance_counter() {}

    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();
    }
  }

  // namespace sine

  // namespace server

  // namespace performance_counters

  // namespace sine

  // namespace server

  // namespace performance_counters
```

(continues on next page)
```cpp
/// 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();

protected:
    bool evaluate();

private:
    typedef hpx::lcos::local::spinlock mutex_type;
    mutable mutex_type mtx_;
    double current_value_;  
    std::uint64_t evaluated_at_;  
    hpx::util::interval_timer timer_;  
};
```
### Table 2.27: AGAS performance counters

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Counter instance formatting</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/agas/count/&lt;agas_service&gt;</td>
<td>&lt;agas_instance&gt;/total where: &lt;agas_instance&gt; is the name of the AGAS service to query.</td>
<td>None</td>
<td>Returns the total number of invocations of the specified AGAS service since its creation.</td>
</tr>
<tr>
<td>where: &lt;agas_service&gt; is one of the following: primary namespace services: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate, begin_migration, end_migration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>component namespace services: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locality namespace services: free, localities, num_localities, num_threads, resolve_locality, resolved_localities</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbol namespace services: bind, resolve, unbind, iterate_names, on_symbol_namespace_event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/agas/&lt;agas_service_category&gt;/count</td>
<td>&lt;agas_instance&gt;/total where: &lt;agas_instance&gt; is the name of the AGAS service to query.</td>
<td>None</td>
<td>Returns the overall total number of invocations of all AGAS services provided by the given AGAS service category since its creation.</td>
</tr>
<tr>
<td>where: &lt;agas_service_category&gt; is one of the following: primary, locality, component or symbol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AGAS services are available on all localities, whereas all other AGAS services are available on locality#0 only.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>agas/time/&lt;agas_service&gt;</td>
<td>&lt;agas_instance&gt;/total where: &lt;agas_instance&gt; is the name of the AGAS service to query.</td>
<td>None</td>
<td>Returns the overall execution time of the specified AGAS service since its creation (in nanoseconds).</td>
</tr>
<tr>
<td>where: &lt;agas_service&gt; is one of the following: primary namespace services: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate, begin_migration, end_migration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>component namespace services: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locality namespace services: free, localities, num_localities, num_threads, resolve_locality, resolved_localities</td>
<td>symbol AGAS service components live on all localities, whereas all other AGAS services are available on locality#0 only.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2.28: Parcel layer performance counters

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Counter instance formatting</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/data/count/</code></td>
<td><code>locality**/total</code></td>
<td>Returns the overall number of raw (uncompressed) bytes sent or received (see <code>&lt;operation&gt;</code>, e.g. <code>sent</code> or <code>received</code>) for the specified <code>&lt;connection_type&gt;</code>. The performance counters for the connection type <code>mpi</code> are available only if the compile time constant <code>HPX_HAVE_PARCELPORT_MPI</code> was defined while compiling the <code>HPX</code> core library (which is not defined by default, the corresponding cmake configuration constant is <code>HPX_WITH_PARCELPORT_MPI</code>). Please see <code>CMake variables used to configure HPX</code> for more details.</td>
<td>None</td>
</tr>
<tr>
<td><code>/data/time/</code></td>
<td><code>locality**/total</code></td>
<td>Returns the total time (in nanoseconds) between the start of each asynchronous transmission operation and the end of the corresponding operation for the specified <code>&lt;connection_type&gt;</code> the given <code>locality</code> (see <code>&lt;operation&gt;</code>, e.g. <code>sent</code> or <code>received</code>). The performance counters for the connection type <code>mpi</code> are available only if the compile time constant <code>HPX_HAVE_PARCELPORT_MPI</code> was defined while compiling the <code>HPX</code> core library (which is not defined by default, the corresponding cmake configuration constant is <code>HPX_WITH_PARCELPORT_MPI</code>). Please see <code>CMake variables used to configure HPX</code> for more details.</td>
<td>None</td>
</tr>
<tr>
<td><code>/serialize/count/</code></td>
<td><code>locality**/total</code></td>
<td>Returns the overall number of bytes transferred (see <code>&lt;operation&gt;</code>, e.g. <code>sent</code> or <code>received</code> possibly compressed) for the specified <code>&lt;connection_type&gt;</code> by the given <code>locality</code>. The performance counters for the connection type <code>mpi</code> are available only if the compile time constant <code>HPX_HAVE_PARCELPORT_MPI</code> was defined while compiling the <code>HPX</code> core library (which is not defined by default, the corresponding cmake configuration constant is <code>HPX_WITH_PARCELPORT_MPI</code>). Please see <code>CMake variables used to configure HPX</code> for more details.</td>
<td>If the configure-time option <code>-DHPX_WITH_PARCELPORT_ACTION</code> 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.</td>
</tr>
<tr>
<td><code>/serialize/time/</code></td>
<td><code>locality**/total</code></td>
<td>Returns the overall time spent performing outgoing data serialization for the specified <code>&lt;connection_type&gt;</code> on the given <code>locality</code> (see <code>&lt;operation&gt;</code>, e.g. <code>sent</code> or <code>received</code>). The performance counters for the connection type <code>mpi</code> are available only if the compile time constant <code>HPX_HAVE_PARCELPORT_MPI</code> was defined while compiling the <code>HPX</code> core library (which is not defined by default, the corresponding cmake configuration constant is <code>HPX_WITH_PARCELPORT_MPI</code>). Please see <code>CMake variables used to configure HPX</code> for more details.</td>
<td>If the configure-time option <code>-DHPX_WITH_PARCELPORT_ACTION</code> 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.</td>
</tr>
</tbody>
</table>
Table 2.29: Thread manager performance counters

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Counter instance formatting</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/count/cumulative</td>
<td>locality#<em>/total or locality#</em>/worker-thread#* or locality#<em>/pool#</em>/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 * 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.</td>
<td>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).</td>
<td>None</td>
</tr>
</tbody>
</table>

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>/threads/time/average</th>
<th>locality#<em>/total or locality#</em>/worker-thread#* or locality#<em>/pool#</em>/worker-thread#* where:</th>
<th>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].</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>locality#* is defining the locality for which the average time spent executing one HPX-thread should be queried for. The locality id (given by <em>) is a (zero based) number identifying the locality. pool#</em> 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.</td>
<td>None</td>
</tr>
</tbody>
</table>

continues on next page
Table 2.29 – continued from previous page

| locality#*/total or locality#*/worker-thread#*/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 *) 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. | 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]. | None |
Table 2.29 – continued from previous page

|/threads/count/cumulative-phases| 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 *) 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. | 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]. | None |

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/time/average-phase</td>
<td>Returns the average time spent executing one <strong>HPX</strong>-thread phase (invocation) on the given <strong>locality</strong> since application start. If the instance name is <code>total</code> the counter returns the average time spent executing one <strong>HPX</strong>-thread phase (invocation) for all worker threads (cores) on that <strong>locality</strong>. If the instance name is <code>worker-thread#*</code> the counter will return the average time spent executing one <strong>HPX</strong>-thread phase for all worker threads separately. This counter is available only if the configuration time constants <strong>HPX</strong>_WITH_THREAD_CUMULATIVE_COUNTS (default: <strong>ON</strong>) and <strong>HPX</strong>_WITH_THREAD_IDLE_RATES are set to <strong>ON</strong> (default: <strong>OFF</strong>). The unit of measure for this counter is nanosecond [ns].</td>
<td></td>
</tr>
</tbody>
</table>

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 `*`) 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.

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>/threads/time/average-phase-overhead</th>
<th>locality#<em>/total or locality#</em>/worker-thread#* or locality#<em>/pool#</em>/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 * 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. 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].</th>
<th>None</th>
</tr>
</thead>
</table>

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>/threads/time/overall</th>
<th>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 *) 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.</th>
<th>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].</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/time/overall</td>
<td>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 *) 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.</td>
<td>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].</td>
<td>None</td>
</tr>
</tbody>
</table>

continues on next page
Table 2.29 – continued from previous page

| /threads/time/cumulative | 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 * 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. | 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). | None |

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Returns the overall overhead time incurred executing all HPX-threads on the given locality since application start. If the instance name is <code>total</code> 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 <code>worker-thread</code> 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 <code>HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS</code> (default: ON) and <code>HPX_THREAD_MAINTAIN_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].</th>
</tr>
</thead>
<tbody>
<tr>
<td>locality#/ 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 * 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 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 <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</td>
</tr>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

continues on next page
Table 2.29 – continued from previous page

| threads/count/ | locality#/total or locality#/worker-thread#/ | 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. | None |
| instantaneous/ | locality#/pool#/worker-thread#/ | |
| <thread-state> | where:<thread-state> is one of the following: all, active, pending, suspended, terminated, staged | |
| 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 * 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. | |

continues on next page
Table 2.29 – continued from previous page

| threads/wait-time/ | locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* | 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]. | None |
| <thread-state> | where: <thread-state> is one of the following: pending staged | | |
| 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. | | |

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Threads/Idle-Rate</th>
<th>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).</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/idle-rate</td>
<td>locality#<em>/total or locality#</em>/worker-thread#* or locality#<em>/pool#</em>/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 * 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.</td>
<td>continues on next page</td>
</tr>
</tbody>
</table>
Table 2.29 – continued from previous page

| /threads/creation-idle-rate | locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* | 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. | None |

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 * 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.

continues on next page
| Threads/ |
|-----------------|---------------------------------|-----------------|
| cleanup-idle-rate | locality#*/total               | 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. |
|                  | or locality#*/                 | None            |
|                  | 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 * 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 |
|                  | 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.         |                 |

Continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threadqueue/length</td>
<td>Returns the overall length of all queues for the given worker thread(s) on the given <code>locality</code>.</td>
<td>locality#/total or locality#/worker-thread#/ or locality#/pool#/worker-thread# where: locality# is defining the <code>locality</code> for which the current length of all thread queues in the scheduler for all (or one) worker threads should be queried for. The <code>locality</code> id (given by * is a (zero based) number identifying the <code>locality</code>. 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 <code>--hpx:threads</code>. If no pool-name is specified the counter refers to the ‘default’ pool.</td>
</tr>
<tr>
<td>/threads/count/stack-unbinds</td>
<td>Returns the total number of HPX-thread unbind (madvise) operations performed for the referenced <code>locality</code>. Note that this counter is not available on Windows based platforms.</td>
<td>locality#/total where: * is the <code>locality</code> id of the <code>locality</code> the unbind (madvise) operations should be queried for. The <code>locality</code> id is a (zero based) number identifying the <code>locality</code>.</td>
</tr>
</tbody>
</table>

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/count/stack-recycles</td>
<td>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.</td>
<td>Returns the total number of HPX-thread recycling operations performed.</td>
</tr>
<tr>
<td>/threads/count/stolen-from-pending</td>
<td>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.</td>
<td>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).</td>
</tr>
</tbody>
</table>

continues on next page
Table 2.29 – continued from previous page

| /threads/count/pending-misses | 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 * 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. | 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). | None |

continues on next page
Table 2.29 – continued from previous page

| /threads/count/pending-accesses | locality#*/total 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 *) 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 \(--\text{hpx:threads}\). If no pool-name is specified the counter refers to the ‘default’ pool. | 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). | None |

continues on next page
Table 2.29 – continued from previous page

| /threads/count/stolen-from-staged | 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 *) 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. | 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). | None |

continues on next page
Table 2.29 – continued from previous page

| /threads/count/stolen-to-pending | 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 # 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. | 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). | None |

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/threads/count/stolen-to-staged</code></td>
<td>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 <code>HPX_WITH_THREAD_STEALING_COUNTS</code> is set to ON (default: ON).</td>
</tr>
</tbody>
</table>

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) 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.
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/count/objects</td>
<td>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.</td>
<td>None</td>
</tr>
<tr>
<td>/locality*/total</td>
<td>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.</td>
<td>None</td>
</tr>
<tr>
<td>allocator*/</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>where:</td>
<td></td>
</tr>
<tr>
<td>locality*/ is defining</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>allocator*/ is defining</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>/scheduler/utilization/instantaneous</td>
<td>Returns the total (instantaneous) scheduler utilization. This is the current percentage of scheduler threads executing HPX threads.</td>
<td>Percent</td>
</tr>
<tr>
<td>locality*/total</td>
<td>Returns the total (instantaneous) scheduler utilization. This is the current percentage of scheduler threads executing HPX threads.</td>
<td>Percent</td>
</tr>
<tr>
<td>where:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>locality*/ is defining</td>
<td>The locality for which the current (instantaneous) scheduler utilization queried for. The locality id (given by *) is a (zero based) number identifying the locality.</td>
<td></td>
</tr>
<tr>
<td>continues on next page</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2.29 – continued from previous page

| /threads/  |
| idle-loop-count/ |
| instantaneous | 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 * 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. |
| Returns the current (instantaneous) idle-loop count for the given HPX-worker thread or the accumulated value for all worker threads. | None |

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th><strong>locality</strong>#*</th>
<th><strong>worker-thread</strong>#*</th>
<th><strong>pool</strong>#*</th>
<th><strong>worker-thread</strong>#*</th>
</tr>
</thead>
<tbody>
<tr>
<td>locality#*</td>
<td>worker-thread#*</td>
<td>pool#*</td>
<td>worker-thread#*</td>
</tr>
<tr>
<td>where:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locality#*</td>
<td>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.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pool#*</td>
<td>is defining the pool for which the current value of the idle-loop counter should be queried for.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>worker-thread#*</td>
<td>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.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Returns the current (instantaneous) busy-loop count for the given HPX-worker thread or the accumulated value for all worker threads.

None

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/time/background-work-duration locality#*/total</td>
<td>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 <code>HPX_WITH_BACKGROUND_THREAD_COUNTERS</code> (default: OFF) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].</td>
<td>None</td>
</tr>
<tr>
<td>locality#/worker-thread#*</td>
<td>where: locality#* is defining the locality for which the overall time spent performing background work should be queried for. The locality id (given by <em>) is a (zero based) number identifying the locality. worker-thread#</em> 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 <code>--hpx:threads</code>.</td>
<td></td>
</tr>
<tr>
<td>continues on next page</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2.29 – continued from previous page

| /threads/background-overhead | 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. | 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%. | None |

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Threads/time/ background-send-duration</th>
<th>locality#* / total</th>
<th>Function</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>locality#* / total</td>
<td>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 parcel-port only.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>locality#* / worker-thread#*</td>
<td>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 <em>) is a (zero based) number identifying the locality. worker-thread#</em> 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.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>/threads/time/ background-send-duration</td>
<td>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 parcel-port only.</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Instance Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/background-send-overhead</td>
<td>Returns the background overhead related to sending parcels on the given locality since application start. If the instance name is <code>total</code>, the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is <code>worker-thread#*</code>, the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_WITH_BACKGROUND_THREAD_COUNTERS</code> (default: <code>OFF</code>) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to <code>ON</code> (default: <code>OFF</code>). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcel-port only.</td>
<td>None</td>
</tr>
</tbody>
</table>

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`.

continues on next page
<table>
<thead>
<tr>
<th>Instance Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/time/background-receive-duration/locality#*</td>
<td>Returns the overall time spent performing background work related to receiving parcels on the given locality since application start. If the instance name is <code>total</code>, the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is <code>worker-thread#*</code>, 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 <code>HPX_WITH_BACKGROUND_THREAD_COUNTERS</code> (default: <code>OFF</code>) and <code>HPX_WITH_THREAD_IDLE_RATES</code> are set to <code>ON</code> (default: <code>OFF</code>). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcelport only.</td>
</tr>
<tr>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
Table 2.29 – continued from previous page

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>/threads/background-receive-overhead</td>
<td>Returns the background overhead related to receiving parcels on the given locality since application start. If the instance name is <code>total</code>, the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is <code>worker-thread#*</code>, the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants <code>HPX_WITH_BACKGROUND_THREAD_COUNTERS</code> (default: OFF) and <code>HPX_WITH_THREAD_IDLE_RATES</code> 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 parcel-port only.</td>
<td>None</td>
</tr>
</tbody>
</table>

locality#*/total overlaphead 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`.

None
Table 2.30: General performance counters exposing characteristics of localities

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Counter instance formatting</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/runtime/count/component</td>
<td>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.</td>
<td>Returns the overall number of currently active components of the specified type on the given locality.</td>
<td>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.</td>
</tr>
<tr>
<td>/runtime/count/action-invocation</td>
<td>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.</td>
<td>Returns the overall (local) invocation count of the specified action type on the given locality.</td>
<td>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.</td>
</tr>
<tr>
<td>/runtime/count/remote-action-invocation</td>
<td>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.</td>
<td>Returns the overall (remote) invocation count of the specified action type on the given locality.</td>
<td>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.</td>
</tr>
<tr>
<td>/runtime/uptime</td>
<td>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.</td>
<td>Returns the overall time since application start on the given locality in nanoseconds.</td>
<td>None</td>
</tr>
<tr>
<td>/runtime/memory/virtual</td>
<td>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.</td>
<td>Returns the amount of virtual memory currently allocated by the referenced locality (in bytes).</td>
<td>None</td>
</tr>
<tr>
<td>/runtime/memory/resident</td>
<td>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.</td>
<td>Returns the amount of resident memory currently allocated by the referenced locality (in bytes).</td>
<td>None</td>
</tr>
</tbody>
</table>

2.3. Manual

| /runtime/memory/total                  | locality#/total where:                                                                            | Returns the total available memory for use by the referenced locality (in bytes).                   | None memory for use by the referenced locality.                                                  |
Table 2.31: Performance counters exposing PAPI hardware counters

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Counter instance formatting</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/papi/&lt;papi_event&gt;</td>
<td>locality#/total or locality#/worker-thread#/</td>
<td>This counter 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).</td>
<td>None</td>
</tr>
</tbody>
</table>

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.

This counter 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 2.32: Performance counters for general statistics

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/statistics/counter/name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.</td>
<td>Returns the current average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name).</td>
<td>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.</td>
</tr>
<tr>
<td>/statistics/counter/rolling_average. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.</td>
<td>Returns the current rolling average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name).</td>
<td>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.</td>
</tr>
<tr>
<td>/statistics/counter/rolling_stddev. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.</td>
<td>Returns the current rolling standard deviation (stdev) value calculated based on the values queried from the underlying counter (the one specified as the instance name).</td>
<td>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 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.</td>
</tr>
<tr>
<td>/statistics/counter/rolling_media. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.</td>
<td>Returns the current (statistically estimated) median value calculated based on the values queried from the underlying counter (the one specified as the instance name).</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 2.33: Performance counters for elementary arithmetic operations

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Counter instance formatting</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/arithmetics/add</td>
<td>None</td>
<td>Returns the sum calculated based on the values queried from the underlying counters (the ones specified as the parameters).</td>
<td>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.</td>
</tr>
<tr>
<td>/arithmetics/</td>
<td>None</td>
<td>Returns the difference calculated based on the values queried from the underlying counters (the ones specified as the parameters).</td>
<td>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.</td>
</tr>
<tr>
<td>subtract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/arithmetics/</td>
<td>None</td>
<td>Returns the product calculated based on the values queried from the underlying counters (the ones specified as the parameters).</td>
<td>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.</td>
</tr>
<tr>
<td>multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/arithmetics/</td>
<td>None</td>
<td>Returns the result of division of the values queried from the underlying counters (the ones specified as the parameters).</td>
<td>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.</td>
</tr>
<tr>
<td>divide</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/arithmetics/</td>
<td>None</td>
<td>Returns the average value of all values queried from the underlying counters (the ones specified as the parameters).</td>
<td>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.</td>
</tr>
<tr>
<td>mean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/arithmetics/</td>
<td>None</td>
<td>Returns the standard deviation of all values queried from the underlying counters (the ones specified as the parameters).</td>
<td>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.</td>
</tr>
<tr>
<td>variance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>median</td>
<td>None</td>
<td>Returns the median value of all values queried from the underlying counters (the ones specified as the parameters).</td>
<td>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.</td>
</tr>
</tbody>
</table>

194 Chapter 2. What’s so special about HPX?
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:

```bash
$ ./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:

```bash
$ ./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
```

2.3. Manual 195
Table 2.34: Performance counters tracking parcel coalescing

<table>
<thead>
<tr>
<th>Counter type</th>
<th>Counter instance formatting</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/coalesce/</td>
<td><code>locality## coalesce</code></td>
<td>Returns the number of parcels handled by the message handler associated with the action which is given by the counter parameter.</td>
<td>The action type. This is the string which has been used while registering the action with <code>HPX</code>, e.g. which has been passed as the second parameter to the macro <code>HPX_REGISTER_ACTION</code> or <code>HPX_REGISTER_ACTION_ID</code>.</td>
</tr>
<tr>
<td>/coalesce/</td>
<td><code>locality## coalesce</code></td>
<td>Returns the number of messages generated by the message handler associated with the action which is given by the counter parameter.</td>
<td>The action type. This is the string which has been used while registering the action with <code>HPX</code>, e.g. which has been passed as the second parameter to the macro <code>HPX_REGISTER_ACTION</code> or <code>HPX_REGISTER_ACTION_ID</code>.</td>
</tr>
<tr>
<td>/coalesce/</td>
<td><code>locality## coalesce</code></td>
<td>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.</td>
<td>The action type. This is the string which has been used while registering the action with <code>HPX</code>, e.g. which has been passed as the second parameter to the macro <code>HPX_REGISTER_ACTION</code> or <code>HPX_REGISTER_ACTION_ID</code>.</td>
</tr>
<tr>
<td>/coalesce/</td>
<td><code>locality## coalesce</code></td>
<td>Returns the average time between arriving parcels for the action which is given by the counter parameter.</td>
<td>The action type. This is the string which has been used while registering the action with <code>HPX</code>, e.g. which has been passed as the second parameter to the macro <code>HPX_REGISTER_ACTION</code> or <code>HPX_REGISTER_ACTION_ID</code>.</td>
</tr>
</tbody>
</table>
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\textsuperscript{132}, which is a framework for application profiling using task timers and various performance counters Huck et al.\textsuperscript{135}. It can be added as a git submodule by turning on the option `HPX_WITH_APEX:BOOL` during CMake configuration. TAU\textsuperscript{133} is an optional dependency when using APEX.

To build `HPX` with APEX, add `HPX_WITH_APEX=ON` and, optionally, `TAU_ROOT=$PATH_TO_TAU` to your CMake configuration. In addition, you can override the tag used for APEX with the `HPX_WITH_APEX_TAG` option. Please see the APEX `HPX` documentation\textsuperscript{134} for detailed instructions on using APEX with `HPX`.

References

2.3.10 `HPX` runtime and resources

`HPX` thread scheduling policies

The `HPX` runtime has five thread scheduling policies: local-priority, static-priority, local, static 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 variables used to configure `HPX` for more information).

Priority local scheduling policy (default policy)

- default or invoke using: `--hpx:queuing local-priority-fifo`

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 and will be available always.

\textsuperscript{132} http://uo-oaciss.github.io/apex

\textsuperscript{133} https://www.cs.uoregon.edu/research/tau/home.php

\textsuperscript{134} https://uo-oaciss.github.io/apex/usage/#hpx-louisiana-state-university
This scheduler can be used with two underlying queuing policies (FIFO: first-in-first-out, and LIFO: last-in-first-out). The default is FIFO. In order to use the LIFO policy use the command line option `--hpx:queuing=local-priority-lifo`.

**Static priority scheduling policy**

- **invoke using:** `--hpx:queuing=static-priority` (or `-qs`)
- **flag to turn on for build:** `HPX_THREAD_SCHEDULERS=all` or `HPX_THREAD_SCHEDULERS=static-priority`

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`. 

---

198 Chapter 2. What’s so special about HPX?
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\textsuperscript{136} 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\textsuperscript{137} communication. MPI\textsuperscript{138} 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 \texttt{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 \texttt{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 \texttt{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 \texttt{hpx::init_params}.

To add a thread pool use the \texttt{hpx::resource::partitioner::create_thread_pool} method. If you simply want to use the default scheduler and scheduler options it is enough to call \texttt{rp.create_thread_pool("my-thread-pool")}.

Then, to add resources to the thread pool you can use the \texttt{hpx::resource::partitioner::add_resource} method. The resource partitioner exposes the hardware topology retrieved using Portable Hardware Locality (HWLOC)\textsuperscript{139} 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 not assigned to a thread pool by the time \texttt{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 \texttt{--hpx:print-bind} is useful for checking that the thread pools have been set up the way you expect.

\textsuperscript{136} https://en.wikipedia.org/wiki/Message_Passing_Interface
\textsuperscript{137} https://en.wikipedia.org/wiki/Message_Passing_Interface
\textsuperscript{138} https://en.wikipedia.org/wiki/Message_Passing_Interface
\textsuperscript{139} 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`.

```cpp
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.

```cpp
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.

```cpp
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_firstPu = 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_firstPu)
            {
                skipped_firstPu = 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.

```cpp
void init_resource_partitioner_handler(hpx::resource::partitioner& rp)
{
    rp.create_thread_pool("my-thread-pool");
}
```

(continues on next page)
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:

```cpp
rp.create_thread_pool("my-thread-pool",
    hpx::resource::policies::local_priority_lifo,
    hpx::policies::scheduler_mode(
        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. Manual** | **201**
2.3.11 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) {
    // Print just the essential error information.
    hpx::cout << "caught exception: " << e.what() << "\n";
    // Print all of the available diagnostic information as stored with
    // the exception.
  }
```

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:

```cpp
// 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";
    hpx::cout << "\{os-thread\}: " << hpx::get_error_os_thread(e) << "\n";
    hpx::cout << "\{thread-id\}: " << std::hex << hpx::get_error_thread_id(e) << "\n";
    hpx::cout << "\{thread-description\}: " << hpx::get_error_thread_description(e) << "\n";
    hpx::cout << "\{state\}: " << std::hex << hpx::get_error_state(e) << "\n";
    hpx::cout << "\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:

```cpp
///////////////////////////////////////////////////////////////////////
// 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);
    if (ec) {
        // Print just the essential error information.
        hpx::cout << "returned error: " << ec.get_message() << "\n";
    }
}
```

(continues on next page)
Note: The error information is transferred back to the invocation site even if it is executed on a different locality.

This example shows 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:

```cpp
// Detailed error reporting using error code
{
    // Start with a few printable 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";
        hpx::cout << "{function}: " << hpx::get_error_function_name(ec) << "\n";
        hpx::cout << "{file}: " << hpx::get_error_file_name(ec) << "\n";
        hpx::cout << "{line}: " << hpx::get_error_line_number(ec) << "\n";
        hpx::cout << "{os-thread}: " << hpx::get_error_os_thread(ec) << "\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";
        hpx::cout << "{state}: " << std::hex << hpx::get_error_state(ec) << "\n";
        hpx::cout << "{stack-trace}: " << hpx::get_error_backtrace(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:

```cpp
// Error reporting using lightweight error code
{
    // [lightweight_error_handling_diagnostic_information
    hpx::cout << "Error reporting using an lightweight error code\n"

    // Create a new error_code instance.
    hpx::error_code ec(hpx::lightweight);

    // 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 just the essential error information.
        hpx::cout << "returned error: " << ec.get_message() << "\n"

        // Print all of the available diagnostic information as stored with // the exception.
    }
}
```

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 facilities.

**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`. Historically, HPX also defines `hpx::endl` and `hpx::flush` but those are just aliases for the corresponding standard manipulators.

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:

```cpp
// 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.

// 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" << hpx::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:

```cpp
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.
2.3.12 Troubleshooting

This section contains commonly encountered problems when compiling or using HPX.

**Undefined reference to boost::program_options**

Boost::ProgramOptions is not ABI compatible between all C++ versions and compilers. Because of this you may see linker errors similar to this:

```plaintext
...: undefined reference to `boost::program_options::operator<<(std::ostream&, _
˓→boost::program_options::options_description const&)'
```

if you are not linking to a compatible version of Boost::ProgramOptions. We recommend that you use hpx::program_options, which is part of HPX, as a replacement for boost::program_options (see program_options). Until you have migrated to use hpx::program_options we recommend that you always build Boost\(^{140}\) libraries and HPX with the same compiler and C++ standard.

**Undefined reference to hpx::cout**

You may see an linker error message that looks a bit like this:

```plaintext
hello_world.cpp:(.text+0x5aa): undefined reference to `hpx::cout'
hello_world.cpp:(.text+0x5c3): undefined reference to `hpx::iostreams::flush'
```

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. See Creating HPX projects for more details.

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\(^{141}\) 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 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

\(^{140}\) https://www.boost.org/
\(^{141}\) https://www.pgas.org/
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\(^{142}\), the J-machine's\(^{143}\) 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 Using 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 objects (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.


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 Objects (LCOs)
- 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 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 STEllAR\textsuperscript{144} Group (Systems Technology, Emergent Parallelism, and Algorithm Research) at Louisiana State University (LSU)\textsuperscript{145}’s Center for Computation and Technology (CCT)\textsuperscript{146} to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STEllAR\textsuperscript{147} is a research group focusing on system software solutions and scientific application development for hybrid and many-core hardware architectures.

\begin{footnotesize}
\textsuperscript{144} https://stellar-group.org
\textsuperscript{145} https://www.lsu.edu
\textsuperscript{146} https://www.cct.lsu.edu
\textsuperscript{147} https://stellar-group.org
\end{footnotesize}
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\(^{149}\)—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\(^{150}\)) and weak scaling (see Gustafson’s Law\(^{151}\)). 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:

- **Starvation** occurs when there is insufficient concurrent work available to maintain high utilization of all resources.
- **Latencies** are imposed by the time-distance delay intrinsic to accessing remote resources and services.
- **Overhead** is work required for the management of parallel actions and resources on the critical execution path, which is not necessary in a sequential variant.
- **Waiting 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-holds, PCs, supercomputers, or the cloud, all suffer from the reign of the 4 horsemen: Starvation, Latency, Overhead, and Contention. 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\(^{152}\), as published back in 1945, and later extended by the Harvard architecture\(^{153}\). 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:

---

148 https://stellar-group.org
149 http://en.wikipedia.org/wiki/FLOPS
150 http://en.wikipedia.org/wiki/Amdahl%27s_law
151 http://en.wikipedia.org/wiki/Gustafson%27s_law
• Performance: as previously mentioned, scalability and efficiency are the main criteria people are interested in.

• Fault tolerance: the low expected mean time between failures (MTBF\textsuperscript{154}) 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\textsuperscript{155}, caching memory hierarchies in all modern processors, the constant optimization of existing MPI\textsuperscript{156} implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use GPGPUs\textsuperscript{157} 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.

\textsuperscript{154} http://en.wikipedia.org/wiki/Mean_time_between_failures

\textsuperscript{155} http://en.wikipedia.org/wiki/InfiniBand

\textsuperscript{156} https://en.wikipedia.org/wiki/Message_Passing_Interface

\textsuperscript{157} 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\textsuperscript{158}). 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)\textsuperscript{159}, Microsoft Parallel Patterns Library (PPL)\textsuperscript{160}, Cilk++\textsuperscript{161}, 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\textsuperscript{162}) or processes (MPI\textsuperscript{163}). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP as the system synchronizes the threads used to execute the different iterations in parallel. In MPI 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.

\textsuperscript{158} \url{http://en.wikipedia.org/wiki/Cray_XMT}  
\textsuperscript{159} \url{https://www.threadingbuildingblocks.org/}  
\textsuperscript{160} \url{https://msdn.microsoft.com/en-us/library/dd492418.aspx}  
\textsuperscript{161} \url{https://software.intel.com/en-us/articles/intel-cilk-plus/}  
\textsuperscript{162} \url{https://openmp.org/wp/}  
\textsuperscript{163} \url{https://en.wikipedia.org/wiki/Message_Passing_Interface}  
\textsuperscript{164} \url{https://openmp.org/wp/}  
\textsuperscript{165} \url{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 (Partitioned Global Address Space) designed to overcome this limitation, such as Chapel, X10, UPC, or Fortress. 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++. The first attempts towards solving related problem go back decades as well, a good example is the Linda coordination language. 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 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++), we see better asynchrony, simpler application codes, and improved scaling.
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++\footnote{https://charm.cs.uiuc.edu/} 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 workqueue 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\footnote{https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOnmIJ0fWFkJzE18}
  - Slides\footnote{https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2016}
- Tutorials repository\footnote{https://github.com/STEllAR-GROUP/tutorials}
- STEllAR Group blog posts\footnote{http://stellar-group.org/blog/}

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.

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.

\footnote{https://charm.cs.uiuc.edu/}
\footnote{https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOnmIJ0fWFkJzE18}
\footnote{https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2016}
\footnote{https://github.com/STEllAR-GROUP/tutorials}
\footnote{http://stellar-group.org/blog/}
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 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 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 off 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::hpx_init`, `hpx::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 `modules_async_distributed` with implementations for the local and distributed cases.
See the API reference of this module for more details.

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.

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` 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 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. The act of getting a future from the `cuda_stream_helper` object in this library hides the creation of a cuda stream event and the attachment of this event to the promise that is backing the future returned.

The usage is best illustrated by looking at an example

```cpp
// create a cuda target using device number 0,1,2...
hpx::cuda::experimental::target target(device);
// create a stream helper object
hpx::cuda::experimental::cuda_future_helper helper(device);

// launch a kernel and return a future
auto fn = cuda_trivial_kernel<double>;
double d = 3.1415;
auto f = helper.async(fn, d);

// attach a continuation to the future
f.then([&](hpx::future<void> &&f) {
    std::cout << "trivial kernel completed \n";
}).get();
```

Kernels and CPU work may be freely intermixed/overlapped and synchronized with futures.

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.

CMake variables

HPX_WITH_CUDA - this is a general option that will enable both HPX_WITH_ASYNC_CUDA and HPX_WITH_COMPUTE_CUDA when turned ON.

HPX_WITH_ASYNC_CUDA=ON enables the building of this module which requires only the presence of CUDA on the system and only exposes cuda+fuures support (HPX_WITH_ASYNC_CUDA may be used when HPX_WITH_COMPUTE_CUDA=OFF).

HPX_WITH_COMPUTE_CUDA=ON enables building HPX compute features that allow parallel algorithms to be passed through to the GPU/CUDA backend.

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::apply, hpx::sync` and `hpx::dataflow`.

See the API reference of this module for more details.
async_mpi

The MPI library is intended to simplify the process of integrating MPI 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

```cpp
int MPI_Isend(buf, count, datatype, rank, tag, comm, request);
```

becomes

```cpp
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...

```cpp
// create an executor for MPI dispatch
hpx::mpi::experimental::executor exec(MPI_COMM_WORLD);

// post an asynchronous receive using MPI_Irecv
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,
// call an application specific function
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).

```cpp
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(...);
```

(continues on next page)
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.

### 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.

### command_line_handling_local

TODO: High-level description of the module.

See the API reference of this 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::util::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.

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::util::basic_any`
- `hpx::util::optional`
- `hpx::util::tuple`

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\(^\text{180}\) 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.

\(^\text{180}\) http://wg21.link/p0443
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.

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::util::function`
- `hpx::util::function_ref`
- `hpx::util::unique_function`
- `hpx::util::bind`
- `hpx::util::bind_back`
- `hpx::util::bind_front`
- `hpx::util::deferred_call`
- `hpx::util::invoke`
• `hpx::util::invoke_fused`
• `hpx::util::mem_fn`
• `hpx::util::one_shot`
• `hpx::util::protect`
• `hpx::util::result_of`

See the *API reference* of the module for more details.

### futures

This module defines the `hpx::lcos::future` and `hpx::lcos::shared_future` classes corresponding to the C++ standard library classes `std::future` and `std::shared_future`. Note that the specializations of `hpx::lcos::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.

### ini

TODO: High-level description of the module.

See the *API reference* of this module for more details.
init_runtime_local

TODO: High-level description of the module.
See the API reference of this module for more details.

io_service

This module provides an abstraction over Boost.ASIO, combining multiple `asio::io_context`s into a single pool. `hpx::util::io_service_pool` provides a simple pool of `asio::io_context`s 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.

itt_notify

This module provides support for profiling with Intel VTune\[81\].
See the API reference of this module for more details.

lcos_local

This module provides the following local LCOs:

- `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::lcos::local::packaged_task`
- `hpx::lcos::local::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.

**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.

See the API reference of the module for more details.

**mpi_base**

This module provides helper functionality for detecting MPI 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`
- `HPX_PP_EXPAND`
- `HPX_PP_NARGS`
- `HPX_PP_STRINGIZE`
- `HPX_PP_STRIP_PARENS`

See the *API reference* of the module for more details.

**program_options**

The module `program_options` is a direct fork of the Boost.ProgramOptions library (Boost V1.70.0). For more information about this library please see [here](https://www.boost.org/doc/libs/1_70_0/doc/html/program_options.html). In order to be included as an HPX module, the Boost.ProgramOptions 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.ProgramOptions (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.ProgramOptions 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. This differs from P1393 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.

---

182 https://www.boost.org/doc/libs/1_70_0/doc/html/program_options.html
183 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0220r0.pdf
184 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p01393r0.html
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.

- **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.
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.

**runtime_local**

TODO: High-level description of the library.

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\footnote{https://www.boost.org/doc/libs/1_72_0/libs/serialization/doc/index.html} 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.

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 which should be used rather than the C++ standard ones in HPX threads:

- \texttt{hpx::lcos::local::barrier}
- \texttt{hpx::lcos::local::condition_variable}
- \texttt{hpx::lcos::local::counting_semaphore}
- \texttt{hpx::lcos::local::event}
- \texttt{hpx::lcos::local::latch}
- \texttt{hpx::lcos::local::mutex}
- \texttt{hpx::lcos::local::no_mutex}
- \texttt{hpx::lcos::local::once_flag}
- \texttt{hpx::lcos::local::recursive_mutex}
- \texttt{hpx::lcos::local::shared_mutex}
- \texttt{hpx::lcos::local::sliding_semaphore}
• `hpx::lcos::local::spinlock (std::mutex compatible spinlock)`
• `hpx::lcos::local::spinlock_no_backoff (boost::mutex compatible spinlock)`
• `hpx::lcos::local::spinlock_pool`

See `lcos_local`, `async_combinators`, and `async` for higher level synchronization facilities.
See the `API reference` of this module for more details.

**tag_invoke**

TODO: High-level description of the module.
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)

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.

\textsuperscript{186} https://www.open-mpi.org/projects/hwloc/
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.

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

This module contains functionality for asynchronously launching work on remote localities: `hpx::async`, `hpx::apply`. 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`, checkpoints 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:

```cpp
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:

```cpp
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 \texttt{vec} and its five elements. \texttt{prepare_checkpoint} takes arbitrary data containers (same as for \texttt{save_checkpoint}), such as \texttt{int}, \texttt{double}, \texttt{float}, \texttt{vector}, and \texttt{future}, and calculates the necessary buffer space for the checkpoint that would be created if \texttt{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:

```cpp
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 \texttt{save_checkpoint}. This is accomplished by passing a launch policy as the first argument. It is important to note that passing \texttt{hpx::launch::sync} will cause \texttt{save_checkpoint} to return a checkpoint instead of a future to a checkpoint. All other policies passed to \texttt{save_checkpoint} will return a future to a checkpoint.

Sometimes checkpoints must be declared before they are used. \texttt{save_checkpoint} allows users to move pre-created checkpoints 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:

```cpp
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 \texttt{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:

```cpp
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 \texttt{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. \texttt{HPX} has two solutions for these issues: stream operator overloads and access iterators.

\texttt{HPX} contains two stream overloads, \texttt{operator\textordmashtext{\textless\textless}} and \texttt{operator\textordmashtext{\textgreater\textgreater}}, to stream data out of and into \texttt{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;
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:

```cpp
// 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:

```cpp
// 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::lcos::barrier`: distributed barrier.
- `hpx::lcos::fold`: performs a fold with a given action on all given global identifiers.
- `hpx::lcos::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. The compute_cuda for extensions to CUDA programmable GPU devices.

See the API reference of the module for more details.

compute_cuda

This module extends the compute module to handle CUDA programmable GPU devices.

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::distribution_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 LCOs. 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.

distribution

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.

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.

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

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.

**Header hpx/algorithm.hpp**

This header includes Header hpx/local/algorithm.hpp and contains overloads of the algorithms for segmented iterators.

**Header hpx/local/algorithm.hpp**


**Classes**

- `hpx::parallel::v2::reduction`
- `hpx::parallel::v2::induction`

**Functions**

- `hpx::adjacent_find`
- `hpx::all_of`
- `hpx::any_of`
- `hpx::copy`
- `hpx::copy_if`
- `hpx::copy_n`
- `hpx::count`
- `hpx::count_if`
- `hpx::ends_with`
- `hpx::equal`
- `hpx::fill`
- `hpx::fill_n`
- `hpx::find`
- `hpx::find_end`
- `hpx::find_first_of`
- `hpx::find_if`
- `hpx::find_if_not`
- `hpx::for_each`
• hpx::for_each_n
• hpx::generate
• hpx::generate_n
• hpx::includes
• hpx::inplace_merge
• hpx::is_heap
• hpx::is_heap_until
• hpx::is_partitioned
• hpx::is_sorted
• hpx::is_sorted_until
• hpx::lexicographical_compare
• hpx::make_heap
• hpx::parallel::v1::max_element
• hpx::merge
• hpx::parallel::v1::min_element
• hpx::parallel::v1::minmax_element
• hpx::parallel::v1::mismatch
• hpx::move
• hpx::none_of
• hpx::nth_element
• hpx::partial_sort
• hpx::partial_sort_copy
• hpx::partition
• hpx::partition_copy
• hpx::remove
• hpx::remove_copy
• hpx::remove_copy_if
• hpx::remove_if
• hpx::replace
• hpx::replace_copy
• hpx::replace_copy_if
• hpx::replace_if
• hpx::reverse
• hpx::reverse_copy
• hpx::rotate
• hpx::rotate_copy
• `hpx::search`
• `hpx::search_n`
• `hpx::set_difference`
• `hpx::set_intersection`
• `hpx::set_symmetric_difference`
• `hpx::set_union`
• `hpx::shift_left`
• `hpx::shift_right`
• `hpx::sort`
• `hpx::stable_partition`
• `hpx::stable_sort`
• `hpx::starts_with`
• `hpx::swap_ranges`
• `hpx::transform`
• `hpx::unique`
• `hpx::unique_copy`
• `hpx::for_loop`
• `hpx::for_loop_strided`
• `hpx::for_loop_n`
• `hpx::for_loop_n_strided`
• `hpx::ranges::adjacent_find`
• `hpx::ranges::all_of`
• `hpx::ranges::any_of`
• `hpx::ranges::copy`
• `hpx::ranges::copy_if`
• `hpx::ranges::copy_n`
• `hpx::ranges::count`
• `hpx::ranges::count_if`
• `hpx::ranges::ends_with`
• `hpx::ranges::equal`
• `hpx::ranges::fill`
• `hpx::ranges::fill_n`
• `hpx::ranges::find`
• `hpx::ranges::find_end`
• `hpx::ranges::find_first_of`
• `hpx::ranges::find_if`
• hpx::ranges::find_if_not
• hpx::ranges::for_each
• hpx::ranges::for_each_n
• hpx::ranges::generate
• hpx::ranges::generate_n
• hpx::ranges::includes
• hpx::ranges::inplace_merge
• hpx::ranges::is_heap
• hpx::ranges::is_heap_until
• hpx::ranges::is_partitioned
• hpx::ranges::is_sorted
• hpx::ranges::is_sorted_until
• hpx::ranges::make_heap
• hpx::ranges::merge
• hpx::ranges::move
• hpx::ranges::none_of
• hpx::ranges::nth_element
• hpx::ranges::partial_sort
• hpx::ranges::partial_sort_copy
• hpx::ranges::partition
• hpx::ranges::partition_copy
• hpx::ranges::set_difference
• hpx::ranges::set_intersection
• hpx::ranges::set_symmetric_difference
• hpx::ranges::set_union
• hpx::ranges::shift_left
• hpx::ranges::shift_right
• hpx::ranges::sort
• hpx::ranges::stable_partition
• hpx::ranges::stable_sort
• hpx::ranges::starts_with
• hpx::ranges::swap_ranges
• hpx::ranges::unique
• hpx::ranges::unique_copy
• hpx::ranges::for_loop
• hpx::ranges::for_loop_strided
**Header hpx/any.hpp**

This header includes *Header hpx/local/any.hpp*.

**Header hpx/local/any.hpp**

Corresponds to the C++ standard library header any\(^{188}\). *hpx::any* is compatible with *std::any*.

**Classes**

- *hpx::any*
- *hpx::any_nonser*
- *hpx::bad_any_cast*
- *hpx::unique_any_nonser*

**Functions**

- *hpx::any_cast*
- *hpx::make_any*
- *hpx::make_any_nonser*
- *hpx::make_unique_any_nonser*

**Header hpx/assert.hpp**

Corresponds to the C++ standard library header cassert\(^{189}\). *HPX_ASSERT* is the *HPX* equivalent to *assert* in cassert. *HPX_ASSERT* can also be used in CUDA device code.

**Macros**

- *HPX_ASSERT*
- *HPX_ASSERT_MSG*

**Header hpx/barrier.hpp**

This header includes *Header hpx/local/barrier.hpp* 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

- `hpx::lcos::barrier`

Header `hpx/local/barrier.hpp`

Corresponds to the C++ standard library header `barrier`\(^{190}\).

Classes

- `hpx::lcos::local::cpp20_barrier`

Header `hpx/channel.hpp`

This header includes `Header hpx/local/channel.hpp` and contains a distributed channel implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

- `hpx::lcos::channel`

Header `hpx/local/channel.hpp`

Contains a local channel implementation.

Classes

- `hpx::lcos::local::channel`

Header `hpx/chrono.hpp`

This header includes `Header hpx/local/chrono.hpp`.

Header `hpx/local/chrono.hpp`

Corresponds to the C++ standard library header `chrono`\(^{191}\). The following replacements and extensions are provided compared to `chrono`\(^{192}\). The classes below are also available in the `hpx::chrono` namespace, not in the top-level `hpx` namespace.


Classes

- `hpx::chrono::high_resolution_clock`
- `hpx::chrono::high_resolution_timer`
- `hpx::chrono::steady_time_point`

Header `hpx/condition_variable.hpp`

This header includes `Header hpx/local/condition_variable.hpp`.

Header `hpx/local/condition_variable.hpp`

Corresponds to the C++ standard library header `condition_variable`\(^{193}\).

Classes

- `hpx::lcos::local::condition_variable`
- `hpx::lcos::local::condition_variable_any`
- `hpx::lcos::local::cv_status`

Header `hpx/exception.hpp`

This header includes `Header hpx/local/exception.hpp`.

Header `hpx/local/exception.hpp`

Corresponds to the C++ standard library header `exception`\(^{194}\). `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.

Macros

- `HPX_THROW_EXCEPTION`

Classes

- `hpx::exception`


Header `hpx/execution.hpp`

This header includes `Header hpx/local/execution.hpp`.

Header `hpx/local/execution.hpp`

Corresponds to the C++ standard library header `execution`\(^{195}\). 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

- `hpx::execution::seq`
- `hpx::execution::par`
- `hpx::execution::par_unseq`
- `hpx::execution::task`

Classes

- `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::auto_chunk_size`
- `hpx::execution::dynamic_chunk_size`
- `hpx::execution::guided_chunk_size`
- `hpx::execution::persistent_auto_chunk_size`
- `hpx::execution::static_chunk_size`

Header `hpx/functional.hpp`

This header includes `Header hpx/local/functional.hpp`.

Header `hpx/local/functional.hpp`

Corresponds to the C++ standard library header `functional`. `hpx::util::function` is a more efficient and serializable replacement for `std::function`.

Constants

The following constants are also available in `hpx::placeholders`, not the top-level `hpx` namespace.

- `hpx::util::placeholders::_1`
- `hpx::util::placeholders::_2`
- ...
- `hpx::util::placeholders::_9`

Classes

- `hpx::util::function`
- `hpx::util::function_nonser`
- `hpx::util::function_ref`
- `hpx::util::unique_function`
- `hpx::util::unique_function_nonser`
- `hpx::traits::is_bind_expression`
- `hpx::traits::is_placeholder`

Functions

- `hpx::util::bind`
- `hpx::util::bind_back`
- `hpx::util::bind_front`
- `hpx::util::invoke`
- `hpx::util::invoke_fused`
- `hpx::util::mem_fn`

Header `hpx/future.hpp`

This header includes `hpx/local/future.hpp` and contains overloads of `hpx::async`, `hpx::apply`, `hpx::sync`, and `hpx::dataflow` that can be used with actions. See *Action invocation* for more information about invoking actions.

**Note:** The alias from `hpx::promise` to `hpx::lcos::promise` is deprecated and will be removed in a future release. The alias `hpx::distributed::promise` should be used in new applications.

### Classes

- `hpx::lcos::promise`

### Functions

- `hpx::async`
- `hpx::apply`
- `hpx::sync`
- `hpx::dataflow`

Header `hpx/local/future.hpp`

Corresponds to the C++ standard library header `future`\(^{197}\). See *Extended facilities for futures* for more information about extensions to futures compared to the C++ standard library.

**Note:** All names except `hpx::lcos::local::promise` are also available in the top-level `hpx` namespace. `hpx::promise` refers to `hpx::lcos::promise`, a distributed variant of `hpx::lcos::local::promise`, but will eventually refer to `hpx::lcos::local::promise` after a deprecation period.

### Classes

- `hpx::lcos::future`
- `hpx::lcos::shared_future`
- `hpx::lcos::local::promise`
- `hpx::launch`

Functions

- `hpx::lcos::make_future`
- `hpx::lcos::make_shared_future`
- `hpx::lcos::make_ready_future`
- `hpx::async`
- `hpx::apply`
- `hpx::sync`
- `hpx::dataflow`
- `hpx::when_all`
- `hpx::when_any`
- `hpx::when_some`
- `hpx::when_each`
- `hpx::wait_all`
- `hpx::wait_any`
- `hpx::wait_some`
- `hpx::wait_each`

Examples

```
#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::lcos::local::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();
```

(continues on next page)
HPX_ASSERT(f5.is_ready());

// Fire-and-forget
hpx::apply([]() {
    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;
    })
    .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;
}
Header \texttt{hpx/init.hpp}

This header contains functionality for starting, stopping, suspending, and resuming the \textit{HPX} runtime. This is the main way to explicitly start the \textit{HPX} runtime. See \textit{Starting the HPX runtime} for more details on starting the \textit{HPX} runtime.

\textbf{Classes}

- \texttt{hpx::init\_params}
- \texttt{hpx::runtime\_mode}

\textbf{Functions}

- \texttt{hpx::init}
- \texttt{hpx::start}
- \texttt{hpx::finalize}
- \texttt{hpx::disconnect}
- \texttt{hpx::suspend}
- \texttt{hpx::resume}

Header \texttt{hpx/latch.hpp}

This header includes \texttt{Header hpx/local/latch.hpp} and contains a distributed latch implementation. This functionality is also exposed through the \texttt{hpx::distributed} namespace. The name in \texttt{hpx::distributed} should be preferred.

\textbf{Classes}

- \texttt{hpx::lcos::latch}

Header \texttt{hpx/local/latch.hpp}

Corresponds to the C++ standard library header \texttt{latch}\textsuperscript{198}.

\textbf{Classes}

- \texttt{hpx::lcos::local::cpp20\_latch}

\textsuperscript{198} \url{http://en.cppreference.com/w/cpp/header/latch}
Header `hpx/mutex.hpp`

This header includes `Header hpx/local/mutex.hpp`.

Header `hpx/local/mutex.hpp`

Corresponds to the C++ standard library header `mutex`\(^{199}\).

Classes

- `hpx::lcos::local::mutex`
- `hpx::lcos::local::no_mutex`
- `hpx::lcos::local::once_flag`
- `hpx::lcos::local::recursive_mutex`
- `hpx::lcos::local::spinlock`
- `hpx::lcos::local::timed_mutex`
- `hpx::lcos::local::unlock_guard`

Functions

- `hpx::lcos::local::call_once`

Header `hpx/memory.hpp`

This header includes `Header hpx/local/memory.hpp`.

Header `hpx/local/memory.hpp`

Corresponds to the C++ standard library header `memory`\(^{200}\). It contains parallel versions of the copy, fill, move, and construct helper functions in `memory`\(^{201}\). See *Using parallel algorithms* for more information about the parallel algorithms.

Functions

- `hpx::uninitialized_copy`
- `hpx::uninitialized_copy_n`
- `hpx::uninitialized_default_construct`
- `hpx::uninitialized_default_construct_n`
- `hpx::uninitialized_fill`
- `hpx::uninitialized_fill_n`


**HPX Documentation, master**

- `hpx::uninitialized_move`
- `hpx::uninitialized_move_n`
- `hpx::uninitialized_value_construct`
- `hpx::uninitialized_value_construct_n`
- `hpx::ranges::uninitialized_copy`
- `hpx::ranges::uninitialized_copy_n`
- `hpx::ranges::uninitialized_default_construct`
- `hpx::ranges::uninitialized_default_construct_n`
- `hpx::ranges::uninitialized_fill`
- `hpx::ranges::uninitialized_fill_n`
- `hpx::ranges::uninitialized_move`
- `hpx::ranges::uninitialized_move_n`
- `hpx::ranges::uninitialized_value_construct`
- `hpx::ranges::uninitialized_value_construct_n`

**Header hpx/numeric.hpp**

This header includes `hpx/local/numeric.hpp`.

**Header hpx/local/numeric.hpp**

Corresponds to the C++ standard library header `numeric`. See *Using parallel algorithms* for more information about the parallel algorithms.

**Functions**

- `hpx::adjacent_difference`
- `hpx::exclusive_scan`
- `hpx::inclusive_scan`
- `hpx::reduce`
- `hpx::transform_exclusive_scan`
- `hpx::transform_inclusive_scan`
- `hpx::transform_reduce`
- `hpx::ranges::exclusive_scan`
- `hpx::ranges::inclusive_scan`
- `hpx::ranges::transform_exclusive_scan`
- `hpx::ranges::transform_inclusive_scan`
**Header hpx/optional.hpp**

This header includes *Header hpx/local/optional.hpp*.

**Header hpx/local/optional.hpp**

Corresponds to the C++ standard library header `optional`. `hpx::util::optional` is compatible with `std::optional`.

**Constants**

- `hpx::util::nullopt`

**Classes**

- `hpx::util::optional`
- `hpx::util::nullopt_t`
- `hpx::util::bad_optional_access`

**Functions**

- `hpx::util::make_optional`

**Header hpx/runtime.hpp**

This header includes *Header hpx/local/runtime.hpp* and contains functions for accessing distributed runtime information.

**Functions**

- `hpx::find_root_locality`
- `hpx::find_all_localities`
- `hpx::find_remote_localities`
- `hpx::find_locality`
- `hpx::get_colocation_id`
- `hpx::get_locality_id`

**Header hpx/local/runtime.hpp**

This header contains functions for accessing local runtime information.

**Typedefs**

- `hpx::startup_function_type`
- `hpx::shutdown_function_type`

**Functions**

- `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`

**Header hpx/system_error.hpp**

This header includes `Header hpx/local/system_error.hpp`.

**Header hpx/local/system_error.hpp**

Corresponds to the C++ standard library header `system_error`\(^{204}\).

**Classes**

- `hpx::error_code`

**Header hpx/task_block.hpp**

This header includes `Header hpx/local/task_black.hpp`.  

Header `hpx/local/task_black.hpp`

Corresponds to the `task_block` feature in N4411\(^{205}\). See *Using task blocks* for more details on using task blocks.

**Classes**

- `hpx::parallel::v2::task_canceled_exception`
- `hpx::parallel::v2::task_block`

**Functions**

- `hpx::parallel::v2::define_task_block`
- `hpx::parallel::v2::define_task_block_restore_thread`

Header `hpx/thread.hpp`

This header includes `Header hpx/local/thread.hpp`.

Header `hpx/local/thread.hpp`

Corresponds to the C++ standard library header `thread`\(^{206}\). 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**

- `hpx::thread`
- `hpx::jthread`

**Functions**

- `hpx::this_thread::yield`
- `hpx::this_thread::get_id`
- `hpx::this_thread::sleep_for`
- `hpx::this_thread::sleep_until`

\(^{205}\) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
\(^{206}\) http://en.cppreference.com/w/cpp/header/thread
Header `hpx/semaphore.hpp`

This header includes `Header hpx/local/semaphore.hpp`.

Header `hpx/local/semaphore.hpp`

Corresponds to the C++ standard library header `semaphore`\(^{207}\).

Classes

- `hpx::lcos::local::cpp20_binary_semaphore`
- `hpx::lcos::local::cpp20_counting_semaphore`

Header `hpx/shared_mutex.hpp`

This header includes `Header hpx/local/shared_mutex.hpp`.

Header `hpx/local/shared_mutex.hpp`

Corresponds to the C++ standard library header `shared_mutex`\(^{208}\).

Classes

- `hpx::lcos::local::shared_mutex`

Header `hpx/stop_token.hpp`

This header includes `Header hpx/local/stop_token.hpp`.

Header `hpx/local/stop_token.hpp`

Corresponds to the C++ standard library header `stop_token`\(^{209}\).

Constants

- `hpx::nostopstate`\(^{207}\) [http://en.cppreference.com/w/cpp/header/semaphore]
- `hpx::nostopstate`\(^{208}\) [http://en.cppreference.com/w/cpp/header/shared_mutex]
- `hpx::nostopstate`\(^{209}\) [http://en.cppreference.com/w/cpp/header/stop_token]
Classes

- `hpx::stop_callback`
- `hpx::stop_source`
- `hpx::stop_token`
- `hpx::nostopstate_t`

Header `hpx/tuple.hpp`

This header includes `Header hpx/local/tuple.hpp`.

Header `hpx/local/tuple.hpp`

Corresponds to the C++ standard library header `tuple`. `hpx::tuple` can be used in CUDA device code, unlike `std::tuple`.

Constants

- `hpx::ignore`

Classes

- `hpx::tuple`
- `hpx::tuple_size`
- `hpx::tuple_element`

Functions

- `hpx::make_tuple`
- `hpx::tie`
- `hpx::forward_as_tuple`
- `hpx::tuple_cat`
- `hpx::get`

---

Header `hpx/type_traits.hpp`

This header includes `Header hpx/local/type_traits.hpp`.

Header `hpx/local/type_traits.hpp`

Corresponds to the C++ standard library header `type_traits`\(^\text{211}\).

Classes

- `hpx::is_invocable`
- `hpx::is_invocable_r`

Header `hpx/unwrap.hpp`

This header includes `Header hpx/local/unwrap.hpp`.

Header `hpx/local/unwrap.hpp`

Contains utilities for unwrapping futures.

Classes

- `hpx::functional::unwrap`
- `hpx::functional::unwrap_n`
- `hpx::functional::unwrap_all`

Functions

- `hpx::unwrap`
- `hpx::unwrap_n`
- `hpx::unwrap_all`
- `hpx::unwrapping`
- `hpx::unwrapping_n`
- `hpx::unwrapping_all`

**Header hpx/version.hpp**

This header provides version information about *HPX*.

**Macros**

- HPX_VERSION_MAJOR
- HPX_VERSION_MINOR
- HPX_VERSION_SUBMINOR
- HPX_VERSION_FULL
- HPX_VERSION_DATE
- HPX_VERSION_TAG
- HPX_AGAS_VERSION

**Functions**

- 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

**Header hpx/wrap_main.hpp**

This header 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.

### 2.8.2 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.
Main HPX library

This lists functionality in the main HPX library that has not been moved to modules yet.

```cpp
namespace hpx
namespace components

Functions

template<typename Component>
future<naming::id_type> migrate_from_storage(naming::id_type const &to_resurrect, naming::id_type const &target = naming::invalid_id)

Migrate the component with the given id from the specified target storage (resurrect the object)

The function migrate_from_storage<Component> will migrate the component referenced by to_resurrect from the storage facility specified where the object is currently stored on. It returns a future referring to the migrated component instance. The component instance is resurrected on the locality specified by target_locality.

Return A future representing the global id of the migrated component instance. This should be the same as to_resurrect.

Parameters

- to_resurrect: [in] The global id of the component to migrate.
- target: [in] The optional locality to resurrect the object on. By default the object is resurrected on the locality it was located on last.

Template Parameters

- The: only template argument specifies the component type of the component to migrate from the given storage facility.

```cpp
template<typename Component>
future<naming::id_type> migrate_to_storage(naming::id_type const &to_migrate, naming::id_type const &target_storage)

Migrate the component with the given id to the specified target storage

The function migrate_to_storage<Component> will migrate the component referenced by to_migrate to the storage facility specified with target_storage. It returns a future referring to the migrated component instance.

Return A future representing the global id of the migrated component instance. This should be the same as migrate_to.

Parameters

- to_migrate: [in] The global id of the component to migrate.
- target_storage: [in] The id of the storage facility to migrate this object to.

Template Parameters

- The: only template argument specifies the component type of the component to migrate to the given storage facility.
template<typename Derived, typename Stub>
Derived migrate_to_storage (client_base<Derived, Stub> const &to_migrate,
hpx::components::component_storage const &target_storage)
Migrate the given component to the specified target storage

The function \textit{migrate\_to\_storage} will migrate the component referenced by \textit{to\_migrate} to the storage facility specified with \textit{target\_storage}. It returns a future referring to the migrated component instance.

\textbf{Return} A client side representation of representing of the migrated component instance. This should be the same as \textit{migrate\_to}.

\textbf{Parameters}

- \textit{to\_migrate}: [in] The client side representation of the component to migrate.
- \textit{target\_storage}: [in] The id of the storage facility to migrate this object to.

\textbf{file migrate\_from\_storage.hpp}

\begin{verbatim}
#include <hpx/config.hpp>
#include "hpx/components_base/traits/is_component.hpp"
#include <hpx/futures/future.hpp>
#include <hpx/naming_base/id_type.hpp>
#include <hpx/components/component_storage/server/migrate_from_storage.hpp>
#include <type_traits>
\end{verbatim}

\textbf{file migrate\_to\_storage.hpp}

\begin{verbatim}
#include <hpx/config.hpp>
#include <hpx/components/client_base.hpp>
#include <hpx/components_base/traits/is_component.hpp>
#include <hpx/futures/future.hpp>
#include <hpx/naming_base/id_type.hpp>
#include <hpx/components/component_storage/server/migrate_to_storage.hpp>
#include <type_traits>
\end{verbatim}

\textbf{file set\_parcel\_write\_handler.hpp}

\begin{verbatim}
#include <hpx/config.hpp>
\end{verbatim}

\textbf{dir /hpx/source/components/component_storage}

\textbf{dir /hpx/source/components/component_storage/include/hpx/components/component_storage}

\textbf{dir /hpx/source/components/component_storage/include/hpx/components}

\textbf{dir /hpx/source/hpx}

\textbf{dir /hpx/source/components/component_storage/include}

\textbf{dir /hpx/source/hpx/runtime}

\textbf{dir /hpx/source}

\textbf{affinity}

The contents of this module can be included with the header \texttt{hpx/modules/affinity.hpp}. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header \texttt{hpx/modules/affinity.hpp}, not the particular header in which the functionality you would like to use is defined. See \textbf{Public API} for a list of names that are part of the public \texttt{HPX} API.

\textbf{namespace hpx}

\textbf{namespace threads}

2.8. API reference
Functions

```cpp
void parse_affinity_options(std::string const & spec, std::vector<mask_type> & affinities, std::size_t used_cores, std::size_t max_cores, std::size_t num_threads, std::vector<std::size_t> & num_pus, bool use_process_mask, error_code & ec = throws)

void parse_affinity_options(std::string const & spec, std::vector<mask_type> & affinities, error_code & ec = throws)
```

algorithms

The contents of this module can be included with the header `hpx/modules/algorithms.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/algorithms.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

```cpp
namespace traits

typedefs

template<typename Source, typename Dest>
using pointer_copy_category_t = typename pointer_copy_category<Source, Dest>::type

template<typename Source, typename Dest>
using pointer_move_category_t = typename pointer_move_category<Source, Dest>::type

template<typename Iterator>
using remove_const_iterator_value_type_t = typename remove_const_iterator_value_type<Iterator>::type

struct general_pointer_tag
    Subclassed by hpx::traits::trivially_copyable_pointer_tag

template<typename Source, typename Dest, typename Enable = void>
struct pointer_copy_category

public types

template<>
using type = typename detail::pointer_copy_category::type

template<typename Source, typename Dest, typename Enable = void>
struct pointer_move_category
```
Public Types

template<>
using type = typename detail::pointer_move_category::type

template<typename Iterator, typename Enable = void>
struct remove_const_iterator_value_type

Public Types

template<>
using type = Iterator

template<typename Iterator>
struct projected_iterator<Iterator, typename std::enable_if<is_segmented_iterator<Iterator>::value>::type>

Public Types

template<>
using local_iterator = typename segmented_iterator_traits::local_iterator

template<>
using type = typename segmented_local_iterator_traits<local_iterator>::local_raw_iterator

template<typename Iterator>
struct projected_iterator<Iterator, typename hpx::util::always_void<
    typename std::decay<
        typename std::decay<
            Iterator
        >::proxy_type
    >::proxy_type

namespace hpx

namespace parallel

namespace traits

Typedefs

template<typename F, typename Iter>
using is_projected_t = typename is_projected<F, Iter>::type

template<typename ExPolicy, typename F, typename ...Projected>
using is_indirect_callable_t = typename is_indirect_callable<
    ExPolicy, F, Projected...>::type
Variables

```cpp
template<typename F, typename Iter> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::parallel::traits::is_projected_v = is_projected<F, Iter>::value

template<typename ExPolicy, typename F, typename... Projected> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::parallel::traits::is_indirect_callable_v = is_indirect_callable<ExPolicy, F, Projected...>::value
```

```cpp
struct projected
```

Public Types

```cpp
template<>
using projector_type = typename std::decay<Proj>::type

template<>
using iterator_type = typename hpx::traits::projected_iterator<Iter>::type
```

namespace traits

```cpp
template<typename T, typename Enable = void>
struct projected_iterator
```

Public Types

```cpp
template<>
using type = typename std::decay::type
```

```cpp
template<typename Iterator>
struct projected_iterator<Iterator, typename hpx::util::always_void<typename std::decay<Iterator>::type>::type>
```

Public Types

```cpp
template<>
using type = typename std::decay<Iterator>::type::proxy_type
```

```cpp
template<typename Iterator>
struct projected_iterator<Iterator, typename std::enable_if<is_segmented_iterator<Iterator>::value>::type>
```

Public Types

```cpp
template<>
using local_iterator = typename segmented_iterator_traits::local_iterator
```

```cpp
template<>
using type = typename segmented_local_iterator_traits<local_iterator>::local_raw_iterator
```

```cpp
template<typename Proj, typename Rng>
struct projected_range<Proj, Rng, typename std::enable_if<hpx::traits::is_range<Rng>::value>::type>
```
Public Types

```cpp
template<>
using projector_type = typename std::decay<Proj>::type

template<>
using iterator_type = typename hpx::traits::range_iterator<Rng>::type
```

namespace hpx

namespace parallel

namespace traits

Variables

```cpp
template<typename Proj, typename Rng>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::parallel::traits::is_projected_range_v=is_projected_range<Proj, Rng>::value
```

```cpp
template<typename Proj, typename Rng>
struct projected_range<Proj, Rng, typename std::enable_if<hpx::traits::is_range<Rng>::value>::type>
```

Public Types

```cpp
template<>
using projector_type = typename std::decay<Proj>::type

template<>
using iterator_type = typename hpx::traits::range_iterator<Rng>::type
```

namespace hpx

namespace traits

```cpp
template<typename Iterator, typename Enable = void>
struct segmented_iterator_traits
```

Public Types

```cpp
typedef std::false_type is_segmented_iterator
```

```cpp
template<typename Iterator, typename Enable = void>
struct segmented_local_iterator_traits
```
Public Types

typedef std::false_type is_segmented_local_iterator
typedef Iterator iterator
typedef Iterator local_iterator
typedef Iterator local_raw_iterator

Public Static Functions

static local_raw_iterator const &local (local_iterator const &it)
static local_iterator const &remote (local_raw_iterator const &it)
static local_raw_iterator local (local_iterator &it)
static local_iterator remote (local_raw_iterator &it)

namespace hpx

namespace lcos

namespace local

Functions

template<typename ExPolicy, typename F, typename ...Args, typename = typename std::enable_if<hpx::parallel::vector<hpx::future<void>>::define_spmd_block (ExPolicy &&policy, std::size_t num_images, F &&f, Args&&... args)>

template<typename ExPolicy, typename F, typename ...Args, typename = typename std::enable_if<hpx::is_async>::define_spmd_block (ExPolicy &&policy, std::size_t num_images, F &&f, Args&&... args)>

void define_spmd_block (std::size_t num_images, F &&f, Args&&... args)

struct spmd_block

#include <spmd_block.hpp> The class spmd_block defines an interface for launching multiple images while giving handles to each image to interact with the remaining images. The define_spmd_block function templates create multiple images of a user-defined function (or lambda) and launches them in a possibly separate thread. A temporary spmd block object is created and diffused to each image. The constraint for the function (or lambda) given to the define_spmd_block function is to accept a spmd_block as first parameter.
Public Functions

```
spmd_block (std::size_t num_images, std::size_t image_id, barrier_type &barrier, table_type &barriers, mutex_type &mtx)
spmd_block (spmd_block&&)
spmd_block (spmd_block const&)
spmd_block &operator= (spmd_block&&)
spmd_block &operator= (spmd_block const&)
std::size_t get_num_images () const
std::size_t this_image () const
void sync_all () const
void sync_images (std::set< std::size_t > const &images) const
void sync_images (std::vector< std::size_t > const &input_images) const
```

```
template<typename Iterator>
std::enable_if<traits::is_input_iterator<Iterator>::value>::type sync_images (Iterator begin, Iterator end)
```

```
template<typename ...I>
std::enable_if<util::all_of<typename std::is_integral<I>::type>::value>::type sync_images (I... i)
```

Private Types

```
using barrier_type = hpx::lcos::local::barrier
using table_type = std::map< std::set< std::size_t >, std::shared_ptr< barrier_type > >
using mutex_type = hpx::lcos::local::mutex
```

Private Members

```
std::size_t num_images_
std::size_t image_id_
std::reference_wrapper< barrier_type > barrier_
std::reference_wrapper< table_type > barriers_
std::reference_wrapper< mutex_type > mtx_
```

namespace parallel
Typedefs

```cpp
using spmd_block = hpx::lcos::local::spmd_block
```

The class `spmd_block` defines an interface for launching multiple images while giving handles to each image to interact with the remaining images. The `define_spmd_block` function templates create multiple images of a user-defined function (or lambda) and launches them in a possibly separate thread. A temporary `spmd_block` object is created and diffused to each image. The constraint for the function (or lambda) given to the `define_spmd_block` function is to accept a `spmd_block` as first parameter.

Functions

```cpp
template<typename ExPolicy, typename F, typename ... Args, typename = std::enable_if<hpx::parallel::execution::is_async_execution_policy<ExPolicy>::value>::type>
std::vector<hpx::future<void>> define_spmd_block(ExPolicy &&policy, std::size_t num_images, F &&f, Args&&... args)
```

```cpp
template<typename ExPolicy, typename F, typename ... Args, typename = std::enable_if<!hpx::is_async_execution_policy<ExPolicy>::value>::type>
void define_spmd_block(ExPolicy &&policy, std::size_t num_images, F &&f, Args&&... args)
```

```cpp
namespace hpx

namespace parallel

Functions

```cpp
template<typename ExPolicy, typename F>
hpx::future<void> 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.

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.

Note It is expected (but not mandated) that `f` will (directly or indirectly) call `tr.run(callable_object)`.

Exceptions

- `An: exception_list`, as specified in Exception Handling.

```cpp
template<typename ExPolicy, typename F>
void define_task_block(ExPolicy &&policy, F &&f)
```
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.

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.

Note It is expected (but not mandated) that f will (directly or indirectly) call tr.run(callable_object).

Exceptions
- An: exception_list, as specified in Exception Handling.

template<typename ExPolicy, typename F>
util::detail::algorithm_result<ExPolicy>::type 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.

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.

Exceptions
- An: exception_list, as specified in Exception Handling.

Note It is expected (but not mandated) that f will (directly or indirectly) call tr.run(callable_object).

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.

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.

Exceptions
  • An: exception_list, as specified in Exception Handling.

Note  It is expected (but not mandated) that \( f \) will (directly or indirectly) call \( tr \).run(callable_object).

namespace v2

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:
```cpp
define_task_block([&](auto& tr) {
  tr.run([&] {
    tr.run([&] {
      tr.run([] { f(); });
    });
  });
  // Error: tr is not active
}

// Nested task block
define_task_block([&](auto& tr) {
  tr.run(f);
  // OK: inner tr is active

  /// ...
});
/// ...
```
**Public Types**

```cpp
template<>
using execution_policy = ExPolicy
```

Refers to the type of the execution policy used to create the `task_block`.

**Public Functions**

```cpp
execution_policy const & get_execution_policy () const
```

Return the execution policy instance used to create this `task_block`.

```cpp
template<typename F, typename ...Ts>
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.

**Exceptions**

- This: function may throw `task_canceled_exception`, as described in Exception Handling.

```cpp
template<typename Executor, typename F, typename ...Ts>
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.

Exceptions
- 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.

```cpp
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.
```

Note The call to `wait` is sequenced before the continuation as if `wait` returns on the same thread.

Exceptions
- 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.

```cpp
ExPolicy &policy ()
Returns a reference to the execution policy used to construct this object.
Precondition: this shall be the active `task_block`.

ExPolicy const &policy () const
Returns a reference to the execution policy used to construct this object.
Precondition: this shall be the active `task_block`.
```

Private Members

```cpp
hpx::execution::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

task_canceled_exception()

namespace hpx

namespace execution

namespace experimental

class task_group

Public Functions

task_group()

~task_group()

template<typename Executor, typename F, typename ...Ts>
void run (Executor &exec, F &f, Ts &... ts)
Spawns a task to compute f() and returns immediately.

template<typename F, typename ...Ts>
void run (F &f, Ts &... ts)

void wait ()
  Waits for all tasks in the group to complete.

void add_exception (std::exception_ptr p)

Private Members

hpx::lcos::local::latch latch_

hpx::exception_list errors_

bool has_arrived_

struct on_exit

Public Functions

on_exit (hpx::lcos::local::latch &l)

~on_exit ()

on_exit (on_exit const &rhs)

on_exit &operator= (on_exit const &rhs)

on_exit (on_exit &rhs)

on_exit &operator= (on_exit &rhs)
Public Members

namespace hpx

Functions

template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
std::enable_if<hpx::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter2>::type>::type

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.

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 an forward iterator.

• FwdIter2: The type of the source iterators used for the output 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 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.

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.

This overload of adjacent_find is available if the user decides to provide their algorithm their own binary predicate 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 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_find algorithm returns an iterator to the last element in the output range.

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.

**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 an forward iterator.
- **FwdIter2**: The type of the source iterators used for the output range (deduced). This iterator type must meet the requirements of an 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:

```cpp
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`.

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.

**Return** 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_find algorithm returns an iterator to the last element in the output range.
Variables

```cpp
std::enable_if<hpx::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter2>::type>::type
```

```cpp
namespace hpx
```

Functions

```cpp
template<typename FwdIter, typename Pred = detail::equal_to>
FwdIter adjacent_find (FwdIter first, FwdIter last, Pred &&pred = Pred())
```

Searches the range \([\text{first}, \text{last})\) for two consecutive identical elements.

**Note** Complexity: Exactly the smaller of \((\text{result} - \text{first}) + 1\) and \((\text{last} - \text{first}) - 1\) application of the predicate where \(\text{result}\) is the value returned

**Return** The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is 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.
- **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:

```cpp
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`.

```cpp
template<typename ExPolicy, typename FwdIter, typename Pred = detail::equal_to>
std::enable_if<hpx::is_execution_policy<ExPolicy>::value, typename util::detail::algorithm_result<ExPolicy, FwdIter>::type>::type
```

Searches the range \([\text{first}, \text{last})\) 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.

**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.
- `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:

```cpp
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`.

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`.

**Return** 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.

```cpp
namespace hpx
```
Functions

template<typename ExPolicy, typename FwdIter, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type none_of(ExPolicy &&policy, FwdIter first, FwdIter 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.

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 none_of requires F to meet the requirements of CopyConstructible.
- Proj: The type of an optional projection function. This defaults to util::projection_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:

  ```cpp
  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.

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.

Return 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.
`util::detail::algorithm_result<ExPolicy, bool>::type any_of (ExPolicy &&policy, FwdIter first, FwdIter last, F &&f, Proj &&proj = Proj())`

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.

**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 a 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`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_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:

  ```cpp
  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.

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.

**Return** 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 FwdIter, typename F, typename Proj = util::projection_identity> util::detail::algorithm_result<ExPolicy, bool>::type all_of (ExPolicy &&policy, FwdIter first, FwdIter last, F &&f, Proj &&proj = Proj())`

Checks if unary predicate `f` returns true for all elements in the range `[first, last)`.

2.8. API reference 281
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.

**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`.

- `Proj`: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
namespace hpx
```
Functions

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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`.

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.

**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.
- **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 `copy` algorithm invoked with an execution policy object of type `sequenced_policy` or `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** 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.

```cpp
hpx::parallel::util::detail::algorithm_result<ExPolicy, 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.

**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 \( \text{first} \) the algorithm will be applied to.

• **dest**: Refers to the beginning of the destination range.

The assignments in the parallel \( \text{copy}_n \) algorithm invoked with an execution policy object of type \( \text{parallel}_\text{policy} \) or \( \text{parallel}_\text{task}_\text{policy} \) are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** The \( \text{copy}_n \) algorithm returns a \( \text{hpx}::\text{future}<\text{FwdIter2}> \) if the execution policy is of type \( \text{sequenced}_\text{task}_\text{policy} \) or \( \text{parallel}_\text{task}_\text{policy} \) and returns \( \text{FwdIter2} \) otherwise. The \( \text{copy} \) 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename F>

hpx::parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type copy_if
(ExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest, Pred &&pred)
```

Copies the elements in the range, defined by \([\text{first}, \text{last})\), to another range beginning at \( \text{dest} \). Copies only the elements for which the predicate \( f \) returns true. The order of the elements that are not removed is preserved.

The assignments in the parallel \( \text{copy}_\text{if} \) algorithm invoked with an execution policy object of type \( \text{sequenced}_\text{policy} \) execute in sequential order in the calling thread.

**Note** Complexity: Performs not more than \( \text{last} - \text{first} \) assignments, exactly \( \text{last} - \text{first} \) 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:

```cpp
template<typename Type>
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`.

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.

**Return** 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` 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.

```cpp
namespace hpx
{

    template<typename ExPolicy, typename FwdIterB, typename FwdIterE, typename T, typename Proj = util::projection_identity>
    util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIterB>::difference_type>::type count(ExPolicy &&policy, FwdIterB first, FwdIterE 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`.

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.

**Template Parameters**

- **ExPolicy**: The type of the execution policy to 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.
- **FwdIterB**: The type of the source begin iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIterE**: The type of the source end iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- **T**: The type of the value to search for (deduced).
- **Proj**: The type of an optional projection function. This defaults to `util::projection_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.

**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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Iter, typename Sent, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<Iter>::difference_type>::type count_if (ExPolicy &&policy, 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.
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.

Return 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<FwdIterB>::difference_type`). The `count` algorithm returns the number of elements satisfying the given criteria.

Template Parameters
- `ExPolicy`: The type of the execution policy to 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 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 `count_if` requires `F` to meet the requirements of `CopyConstructible`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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 `FwdIterB` 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.

```cpp
namespace hpx
```
Functions

```cpp
template<typename ExPolicy, typename FwdIter>
util::detail::algorithm_result<ExPolicy>::type destroy (ExPolicy &&policy, FwdIter first, FwdIter 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.

**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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Size>
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.

**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.

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.

**Return** 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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename InIter1, typename InIter2, typename Pred, typename Proj1, typename Proj2>
bool ends_with(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Pred &&pred, Proj1 &&proj1, 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**
• **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.
• **Proj1**: The type of an optional projection function for the source range. This defaults to `util::projection_identity`
• **Proj2**: The type of an optional projection function for the destination range. This defaults to `util::projection_identity`

**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.
• **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.

**Return** 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, typename Proj1, typename Proj2>

hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy &&policy, FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2, Pred &&pred, Proj1 &&proj1, 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 \textit{ends\_with} algorithm invoked with an execution policy object of type \textit{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Linear: at most \text{min}(N_1, N_2) applications of the predicate and both projections.

\textbf{Template Parameters}

- \textit{ExPolicy}: The type of the execution policy to 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.
- \textit{FwdIter1}: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \textit{FwdIter2}: The type of the begin destination iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- \textit{Pred}: The binary predicate that compares the projected elements.
- \textit{Proj1}: The type of an optional projection function for the source range. This defaults to \textit{util::projection\_identity}
- \textit{Proj2}: The type of an optional projection function for the destination range. This defaults to \textit{util::projection\_identity}

\textbf{Parameters}

- \textit{policy}: The execution policy to use for the scheduling of the iterations.
- \textit{first1}: Refers to the beginning of the source range.
- \textit{last1}: Refers to the end of the source range.
- \textit{first2}: Refers to the beginning of the destination range.
- \textit{last2}: Refers to the end of the destination range.
- \textit{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 \textit{proj1} and \textit{proj2} respectively.
- \textit{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 \textit{is} invoked.
- \textit{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 \textit{is} invoked.

The assignments in the parallel \textit{ends\_with} algorithm invoked with an execution policy object of type \textit{parallel\_policy} or \textit{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \textit{ends\_with} algorithm returns a \textit{hpx::future<bool>} if the execution policy is of type \textit{sequenced\_task\_policy} or \textit{parallel\_task\_policy} and returns \textit{bool} otherwise. The \textit{ends\_with} algorithm returns a boolean with the value true if the second range matches the suffix of the first range, false otherwise.

\begin{flushright}
namespace hpx
\end{flushright}
Functions

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result<ExPolicy, bool>::type 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.

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.

Note Complexity: At most min(last1 - first1, last2 - first2) 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 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:

```cpp
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.

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.
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.

The \emph{equal} algorithm returns a \cpp{hpx::future<bool>} if the execution policy is of type \cpp{sequenced_task_policy} or \cpp{parallel_task_policy} and returns \cpp{bool} otherwise. The \emph{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.

The comparison operations in the parallel \emph{equal} algorithm invoked with an execution policy object of type \cpp{sequenced_policy} execute in sequential order in the calling thread.

Complexity: At most \(last1 - first1\) applications of the predicate \(f\).

**Template Parameters**

- \cpp{ExPolicy}: The type of the execution policy to 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.
- \cpp{FwdIter1}: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- \cpp{FwdIter2}: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- \cpp{Pred}: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of \emph{equal} requires \cpp{Pred} to meet the requirements of \cpp{CopyConstructible}. This defaults to \cpp{std::equal_to<\>()}.

**Parameters**

- \cpp{policy}: The execution policy to use for the scheduling of the iterations.
- \cpp{first1}: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- \cpp{last1}: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- \cpp{first2}: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- \cpp{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:

  ```cpp
  bool pred(const Type1 &a, const Type2 &b);
  ```

The signature does not need to have \cpp{const &}, but the function must not modify the objects passed to it. The types \cpp{Type1} and \cpp{Type2} must be such that objects of types \cpp{FwdIter1} and \cpp{FwdIter2} can be dereferenced and then implicitly converted to \cpp{Type1} and \cpp{Type2} respectively.

The comparison operations in the parallel \emph{equal} algorithm invoked with an execution policy object of type \cpp{parallel_policy} or \cpp{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.
Note The two ranges are considered equal if, for every iterator \( i \) in the range \([\text{first1}, \text{last1})\), \(*i\) equals \(*(\text{first2} + (i - \text{first1})).\) This overload of equal uses operator\(==\) to determine if two elements are equal.

Return The \textit{equal} algorithm returns a \texttt{hpx\::future<bool>} if the execution policy is of type \texttt{sequenced_task_policy} or \texttt{parallel_task_policy} and returns \texttt{bool} otherwise. The \textit{equal} algorithm returns \texttt{true} if the elements in the two ranges are equal, otherwise it returns \texttt{false}.

```cpp
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 \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \texttt{GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, \ldots, *(first + (i - \text{result}) - 1))}.

The reduce operations in the parallel \textit{exclusive_scan} algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: \(O(\text{last} - \text{first})\) applications of the predicate \texttt{std::plus<T>}.

Template Parameters

- \texttt{InIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- \texttt{OutIter}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- \texttt{T}: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \texttt{last}: Refers to the end of the sequence of elements the algorithm will be applied to.
- \texttt{dest}: Refers to the beginning of the destination range.
- \texttt{init}: The initial value for the generalized sum.

The difference between \textit{exclusive_scan} and \textit{inclusive_scan} is that \textit{inclusive_scan} includes the \(i\)th input element in the \(i\)th sum.

Return The \textit{exclusive_scan} algorithm returns \texttt{OutIter}. The \textit{exclusive_scan} algorithm returns the output iterator to the element in the destination range, one past the last element copied.

Note \texttt{GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, \ldots, aN)} is defined as:

- \(a1\) when \(N\) is 1
- \texttt{GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, \ldots, aK)}
  \[- \texttt{GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, \ldots, aN)}\) where \(1 < K+1 = M \leq N\).

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T>
```
Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, \ldots, *(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.

**Note** Complexity: \( O(\text{last} - \text{first}) \) applications of the predicate std::plus\(<\text{T}>\).

**Template Parameters**

- **ExPolicy**: The type of the execution policy to 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.

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 \( i \)th input element in the \( i \)th sum.

**Return** The exclusive_scan algorithm returns a hpx::future\(<\text{FwdIter2}>\) 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.

**Note** GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, \ldots, aN) is defined as:

- \( a1 \) when \( N = 1 \)

- GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, \ldots, aK)

  − GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, \ldots, aN) where \( 1 < K+1 = M <= N \).
**OutIter exclusive_scan** *(InIter first, InIter last, OutIter dest, T init, Op &&op)*

Assigns through each iterator \(i\) in \([\text{result, result + (last - first)})\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, \ldots, *(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.

**Note** Complexity: \(O(last - first)\) applications of the predicate \(op\).

**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:

```cpp
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.

The difference between exclusive_scan and inclusive_scan is that inclusive_scan includes the \(i\)th input element in the \(i\)th sum. If \(op\) is not mathematically associative, the behavior of inclusive_scan may be non-deterministic.

**Return** 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.

**Note** GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aN) is defined as:

- \(a1\) when \(N\) is 1
- \(op\)\(\)\( GENERALIZED\_NONCOMMUTATIVE\_SUM\(\)\( op, a1, \ldots, aK\), \(\)\( GENERALIZED\_NONCOMMUTATIVE\_SUM\(\)\( op, aM, \ldots, aN\)) where \(1 < K+1 = M \leq N\).
Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \texttt{GENERALIZED_NONCOMMUTATIVE\_SUM}(\texttt{binary\_op}, \texttt{init}, \ast\text{first}, \ldots, \ast(\text{first} + (i - \text{result}) - 1))

The reduce operations in the parallel \texttt{exclusive\_scan} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: \( O(\text{last} - \text{first}) \) applications of the predicate \texttt{op}.

\textbf{Template Parameters}

- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{FwdIter1}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{FwdIter2}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{T}: The type of the value to be used as initial (and intermediate) values (deduced).
- \texttt{Op}: The type of the binary function object used for the reduction operation.

\textbf{Parameters}

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \texttt{last}: Refers to the end of the sequence of elements the algorithm will be applied to.
- \texttt{dest}: Refers to the beginning of the destination range.
- \texttt{init}: The initial value for the generalized sum.
- \texttt{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:

\[
\text{Ret } \text{fun}(\text{const Typel } &a, \text{ const Typel } &b);\]

The signature does not need to have \texttt{const\&}, but the function must not modify the objects passed to it. The types \texttt{Typel} and \texttt{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.

The reduce operations in the parallel \texttt{exclusive\_scan} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between \texttt{exclusive\_scan} and \texttt{inclusive\_scan} is that \texttt{inclusive\_scan} includes the \( i \)th input element in the \( i \)th sum. If \texttt{op} is not mathematically associative, the behavior of \texttt{inclusive\_scan} may be non-deterministic.

\textbf{Return} The \texttt{exclusive\_scan} algorithm returns a \texttt{hpx::future<OutIter>} if the execution policy is of type \texttt{sequenced\_task\_policy} or \texttt{parallel\_task\_policy} and returns \texttt{OutIter} otherwise. The \texttt{exclusive\_scan} algorithm returns the output iterator to the element in the destination range, one past the last element copied.

\textbf{Note} \texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\texttt{op}, a1, \ldots, aN) is defined as:
• \(a_1\) when \(N\) is 1

• \(\text{op}(\text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, \ a_1, \ldots, \ a_K), \ \text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, \ a_M, \ldots, \ a_N))\) where \(1 < K+1 = M \leq N\).

namespace hpx

Functions

template<typename ExPolicy, typename FwdIter, typename T>
util::detail::algorithm_result<ExPolicy>::type fill (ExPolicy &&policy, FwdIter first, FwdIter last, T value)
Assigns the given value to the elements in the range \([\text{first}, \text{last})\).

The comparisons in the parallel \texttt{fill} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Performs exactly \(\text{last} - \text{first}\) assignments.

\textbf{Template Parameters}

• \texttt{ExPolicy}: The type of the execution policy to 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.

• \texttt{FwdIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

• \texttt{T}: The type of the value to be assigned (deduced).

Parameters

• \texttt{policy}: The execution policy to use for the scheduling of the iterations.

• \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.

• \texttt{last}: Refers to the end of the sequence of elements the algorithm will be applied to.

• \texttt{value}: The value to be assigned.

The comparisons in the parallel \texttt{fill} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \texttt{fill} algorithm returns a \texttt{hpx::future<void>} if the execution policy is of type \texttt{sequenced\_task\_policy} or \texttt{parallel\_task\_policy} and returns \texttt{difference\_type} otherwise (where \texttt{difference\_type} is defined by \texttt{void}).

template<typename ExPolicy, typename FwdIter, typename Size, typename T>
util::detail::algorithm_result<ExPolicy, FwdIter>::type fill_n (ExPolicy &&policy, FwdIter first, Size count, T value)
Assigns the given value \(\text{value}\) to the first \(\text{count}\) elements in the range beginning at \texttt{first} if \(\text{count} > 0\). Does nothing otherwise.

The comparisons in the parallel \texttt{fill\_n} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Performs exactly \(\text{count}\) assignments, for \(\text{count} > 0\).
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 output 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 \( \text{first} \) the algorithm will be applied to.
- **value**: The value to be assigned.

The comparisons in the parallel \( \text{fill}_n \) algorithm invoked with an execution policy object of type \( \text{parallel}\_policy \) or \( \text{parallel}\_task\_policy \) are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return

The \( \text{fill}_n \) algorithm returns a \( \text{hpx}\_\text{future}\langle\text{void}\rangle \) if the execution policy is of type \( \text{sequenced}\_\text{task}\_policy \) or \( \text{parallel}\_\text{task}\_policy \) and returns \( \text{difference}\_\text{type} \) otherwise (where \( \text{difference}\_\text{type} \) is defined by \( \text{void} \)).

```cpp
namespace hpx
{

Functions

```template<

type ExPolicy, typename FwdIter, typename T>

util::detail::algorithm_result<ExPolicy, FwdIter>::type find(
ExPolicy &&policy, FwdIter first, FwdIter last, T const &val)

Returns the first element in the range \([\text{first}, \text{last})\) that is equal to value

The comparison operations in the parallel \( \text{find} \) algorithm invoked with an execution policy object of type \( \text{sequenced}\_\text{policy} \) execute in sequential order in the calling 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 a 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

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.

Return 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 FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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

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.

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:

```cpp
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.
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.

**Return** 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 FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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

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.

**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.

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.

**Return** 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 algo-
The algorithm returns the first element in the range \([\text{first}, \text{last})\) that does not satisfy the predicate \(f\). If no such element exists that does not satisfy the predicate \(f\), the algorithm returns \(\text{last}\).

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result<ExPolicy, FwdIter1>::type find_end(ExPolicy &&policy, FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2, Pred &&op = Pred())
```

Returns the last subsequence of elements \([\text{first2}, \text{last2})\) found in the range \([\text{first}, \text{last})\) using the given predicate \(f\) to compare elements.

The comparison operations in the parallel \(\text{find\_end}\) algorithm invoked with an execution policy object of type \(\text{sequenced\_policy}\) execute in sequential order in the calling thread.

**Note** Complexity: at most \(S \times (N-S+1)\) comparisons where \(S = \text{distance}(\text{first2}, \text{last2})\) and \(N = \text{distance}(\text{first1}, \text{last1})\).

**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 \(\text{replace}\) requires \(\text{Pred}\) to meet the requirements of \(\text{CopyConstructible}\). This defaults to std::equal_to<>.
- **Proj**: The type of an optional projection function. This defaults to util::projection_identity and is applied to the elements of type dereferenced \(\text{FwdIter1}\) and dereferenced \(\text{FwdIter2}\).

**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 \(\text{true}\) if the elements should be treated as equal. The signature should be equivalent to the following:

```cpp
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 \(\text{Type1}\) and \(\text{Type2}\) must be such that objects of types \(\text{FwdIter1}\) and \(\text{FwdIter2}\) can be dereferenced and then implicitly converted to \(\text{Type1}\) and \(\text{Type2}\) respectively.
• **proj**: Specifies the function (or function object) which will be invoked for each of the elements of type dereferenced `FwdIter1` and dereferenced `FwdIter2` as a projection operation before the function \( f \) is invoked.

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 \( f \).

**Return** 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.

```cpp
template<
    typename ExPolicy, 
    typename FwdIter1, 
    typename FwdIter2, 
    typename Pred = detail::equal_to>
util::detail::algorithm_result<ExPolicy, FwdIter1>::type 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 \( p \) 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.

**Note** Complexity: at most \((S*N)\) comparisons where \( S = \text{distance}(s\_first, s\_last) \) and \( N = \text{distance}(first, last) \).

**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<>`

- **Proj1**: The type of an optional projection function. This defaults to `util::projection_identity` and is applied to the elements of type dereferenced `FwdIter1`.

- **Proj2**: The type of an optional projection function. This defaults to `util::projection_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**: The binary predicate which returns true if the elements should be treated as equal. The signature should be equivalent to the following:

```cpp
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 function op 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 function op is invoked.

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.

**Return** 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. This overload of find_end is available if the user decides to provide the algorithm their own predicate f.

### namespace hpx

#### Functions

```cpp
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.

**Note** Complexity: Applies f exactly last - first times.

If the type of first satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator.

**Return** f.

#### 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 *Move-Constructible*.

**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:

```cpp
<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*.

```cpp
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, void>::type for_each(
    ExPolicy &&policy,
    FwdIter first,
    FwdIter 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.

**Note** Complexity: Applies f exactly last - first times.

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.

**Template Parameters**

• **ExPolicy**: The type of the execution 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 an 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:
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.

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.

Return 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.

Note Complexity: Applies f exactly count times.

If the type of first satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator.

Return first + count for non-negative values of count and first for negative values.

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 Move-Constructible.

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 FwdIter can be dereferenced and then implicitly converted to Type.

template<typename ExPolicy, typename FwdIter, typename Size, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.
If \( f \) returns a result, the result is ignored.

**Note** Complexity: Applies \( f \) exactly \( count \) times.

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.

**Template Parameters**

- `ExPolicy`: The type of the execution 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` 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)`.

```cpp
<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`.

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.

**Return** 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.

```cpp
namespace hpx
```
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.

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:

  \[
  \text{<ignored> pred}(I \text{ const}\& a, \ldots);
  \]

  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.

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:

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.

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.

template<typename ExPolicy, typename I, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type 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.

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.

Template Parameters

- **ExPolicy**: The type of the execution 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:

\[
\texttt{<\textit{ignored}> \mathit{pred}(I \texttt{const}\ a, \texttt{\ldots});}
\]

The signature does not need to have \texttt{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.

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 \( \text{last} - \text{first} \).

The first element in the input sequence is specified by \( \text{first} \). 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:

**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.

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.

**Return** 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.

```cpp
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. \( f \) shall meet the requirements of `MoveConstructible`.

**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, its 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 \([\text{first}, \text{last})\] should expose a signature equivalent to:

```cpp
<ignored> pred(I const& a, ...);
```

The signature does not need to have \( \text{const}\& \). 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.
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:

**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.

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.

```cpp
template<typename ExPolicy, typename I, typename S, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type 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.

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`.

**Template Parameters**

- **ExPolicy**: The type of the execution 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, its 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:

```cpp
<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.

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:

**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.

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.

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.

```cpp
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.

**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, its 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:

  ```cpp
  <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.

**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:

**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.

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.

```cpp
template<typename ExPolicy, typename I, typename Size, typename ...Args>
util::detail::algorithm_result<ExPolicy>::type 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.

**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`.

### Template Parameters
• **ExPolicy**: The type of the execution 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:

```cpp
<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.

**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:

**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.

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.

**Return** 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.
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.

**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, its 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:

  ```cpp
  <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.

  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:

  **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.
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.

```cpp
template<typename ExPolicy, typename I, typename Size, typename S, typename ... Args>
util::detail::algorithm_result<ExPolicy>::type 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.

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.

**Template Parameters**

- **ExPolicy**: The type of the execution 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 \([\text{first}, \text{last})\) should expose a signature equivalent to:

```cpp
<ignored> pred(I const& a, ...);
```

The signature does not need to have \( \text{const}\& \). 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.

**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 \( \text{last} - \text{first} \).
The first element in the input sequence is specified by \textit{first}. Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the \textit{args} parameter pack excluding \textit{f}, an additional argument is passed to each application of \textit{f} as follows:

\textbf{Note} As described in the C++ standard, arithmetic on non-random-access iterators is performed using \texttt{advance} and \texttt{distance}.

\textbf{Note} The order of the elements of the input sequence is important for determining ordinal position of an application of \textit{f}, even though the applications themselves may be unordered.

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 \textit{f} in the input sequence.

\textbf{Complexity:} Applies \textit{f} exactly once for each element of the input sequence.

\textbf{Remarks:} If \textit{f} returns a result, the result is ignored.

\textbf{Return} The \texttt{for\_loop\_n\_strided} algorithm returns a \texttt{hpx::future\<void\>} if the execution policy is of type \texttt{hpx::execution::sequenced\_task\_policy} or \texttt{hpx::execution::parallel\_task\_policy} and returns \texttt{void} otherwise.

\textbf{namespace hpx}

\textbf{namespace parallel}

\textbf{Functions}

\texttt{template<typename T> constexpr 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 \texttt{value} of that type and, optionally, a stride.

For each element in the input range, a looping algorithm over input sequence \textit{S} computes an induction value from an induction variable and ordinal position \texttt{p} within \textit{S} by the formula \(i + p \times \text{stride}\) if a stride was specified or \(i + p\) otherwise. This induction value is passed to the element access function.

If the \texttt{value} argument to \textit{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 \(i + n \times \text{stride}\) to the live-out object upon return, where \(n\) is the number of elements in the input range.

\textbf{Return} This returns an induction object with value type \texttt{T}, initial value \texttt{value}, and (if specified) stride \texttt{stride}. If \texttt{T} is an lvalue of non-const type, \texttt{value} is used as the live-out object for the induction object; otherwise there is no live-out object.

\textbf{Template Parameters}

- \texttt{T}: The value type to be used by the induction object.

\textbf{Parameters}

- \texttt{value: [in]} The initial value to use for the induction object
- \texttt{stride: [in]} The (optional) stride to use for the induction object (default: 1)
namespace hpx

namespace parallel

Functions

template<typename T, typename Op>
constexpr detail::reduction_helper<T, typename std::decay<Op>::type> 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.

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.

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.

Return 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 hpx

Functions

```
template<typename ExPolicy, typename FwdIter, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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`.

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.

**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`.

The assignments in the parallel `generate` algorithm invoked with an execution policy object of type `sequenced_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** 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 FwdIter, typename Size, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.
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.
- **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.

  The assignments in the parallel `generate_n` algorithm invoked with an execution policy object of type `sequenced_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return 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`.

namespace hpx

Functions

```
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::less>
util::detail::algorithm_result<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.

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.

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<>`.

### 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:

```cpp
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.

The comparison operations in the parallel `includes` algorithm invoked with an execution policy object of type ` sequenced_task_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** 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.

```cpp
namespace hpx
```
**Functions**

template<typename InIter, typename OutIter>  
OutIter inclusive_scan(InIter first, InIter last, OutIter dest)  
Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(\(+, \ast \text{first}, \ldots, \ast (\text{first} + (i - \text{result}))\)).

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

**Note** Complexity: \(O(last - first)\) applications of the predicate \(op\).

**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.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the \(i\)th input element in the \(i\)th sum.

**Return** 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.

**Note** GENERALIZED_NONCOMMUTATIVE_SUM(\(+, a_1, \ldots, a_N\)) is defined as:

- \(a_1\) when \(N\) is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(\(+, a_1, \ldots, a_K\)
  
- GENERALIZED_NONCOMMUTATIVE_SUM(\(+, a_M, \ldots, a_N\)) where \(1 < K+1 = M \leq N\).

template<typename ExPolicy, typename FwdIter1, typename FwdIter2>  
util::detail::algorithm_result<ExPolicy, FwdIter2>::type inclusive_scan(ExPolicy &&policy,  
FwdIter1 first, FwdIter1 last, FwdIter2 dest)  
Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(\(+, \ast \text{first}, \ldots, \ast (\text{first} + (i - \text{result}))\)).

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.

**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 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.

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 `i`th input element in the `i`th sum.

**Return** 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.

**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.

```cpp
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.

**Note** Complexity: O(last - first) applications of the predicate `op`.

**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:

```cpp
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.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the ith input element in the ith sum.

**Return** 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.

**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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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)))`.

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.

**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 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:

```cpp
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.

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.

**Return** 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.

**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`.

```cpp
template<typename InIter, typename OutIter, typename T, typename Op>
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.

**Note** Complexity: `O(last - first)` applications of the predicate `op`.

**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:

\[
\text{Ret } \text{fun}(\text{const Type1 } \&a, \text{ const Type1 } \&b);
\]

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.

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.

**Return** 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.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aN)` is defined as:

- a1 when N is 1
- \(\text{op}(\text{GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aK)}, \text{GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \ldots, aN)})\) where 1 < K+1 = M <= N.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T, typename Op>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type inclusive_scan(
  ExPolicy &&policy,
  FwdIter1 first,
  FwdIter1 last,
  FwdIter2 dest,
  T init,
  Op &&op)
```

Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, \ldots, *(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.

**Note** Complexity: \(O(\text{last} - \text{first})\) 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 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).

- **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.

• **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:

```cpp
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.

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.

**Return** 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.

**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.

### namespace hpx

**Functions**

```cpp
#include <hpx/parallel/hpx.h>
```

```cpp
namespace hpx {
    namespace util {
        namespace detail {
            template<typename ExPolicy, typename RandIter, typename Comp = detail::less>
            using algorithm_result = util::algorithm_result<ExPolicy, bool>;
        }
    }
    template<typename ExPolicy, typename RandIter, typename Comp = detail::less>
    util::detail::algorithm_result<ExPolicy, bool>::type is_heap(ExPolicy &policy, RandIter first, RandIter last, Comp &comp = Comp())
    {
        return is_heap(policy, first, last, 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: 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.

- **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.

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.

Return 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 Parameters

• ExPolicy: The type of the execution policy to 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.
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.

**Return** 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.

```cpp
namespace hpx
{

Functions

template<typename FwdIter, typename Pred>
bool is_partitioned (FwdIter first, FwdIter last, Pred &&pred)

Determine if the range [first, last) is partitioned.

Note Complexity: at most (N) predicate evaluations where N = distance(first, last).

**Return** 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 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

```cpp
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`.

```cpp
util::detail::algorithm_result<ExPolicy, bool>::type is_partitioned (ExPolicy &&policy, FwdIter first, FwdIter last, Pred &&pred)

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.

**Note** Complexity: at most (N) predicate evaluations where \( N = \text{distance} \text{(first, last)} \).

**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 algorithm. 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

```cpp
def 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`.

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.

**Return** 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.

```cpp
namespace hpx
{

Functions

template<typename FwdIter, typename Pred = hpx::parallel::v1::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 \( N = \text{distance} \text{(first, last)} \). \( S = \text{number of partitions} \).
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

\[
\text{bool pred(const Type } &a, \text{ const Type } &b);\]

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**.

Return

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.

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.

*Note* Complexity: at most (N+S-1) comparisons where \( N = \text{distance(first, last)} \). \( S = \text{number of partitions} \)

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

```cpp
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`.

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.

**Return** 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.

```cpp
template<typename FwdIter, typename Pred = hpx::parallel::v1::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

```cpp
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`.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Pred = hpx::parallel::v1::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.

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.

**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

```cpp
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`.

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.

**Return** 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.

```cpp
namespace hpx
```
Functions

template<typename InIter1, typename InIter2, typename Pred>
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).

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 Copy-Constructible. 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

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

Return 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 FwdIter1, typename FwdIter2, typename Pred>
util::detail::algorithm_result<ExPolicy, bool>::type 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.

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).

Template Parameters
- ExPolicy: The type of the execution policy to 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 Copy-Constructible. 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

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 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

Return

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.

namespace hpx

Functions

template<typename ExPolicy, typename RndIter, typename Comp>
util::detail::algorithm_result<ExPolicy>::type make_heap(ExPolicy &&policy, RndIter first, RndIter last, Comp &&comp)

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.

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

• 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

```cpp
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.

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.
Return 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<ExPolicy>::type make_heap(ExPolicy &&policy, RndIter first, RndIter last)
    Constructs a max heap in the range [first, last). Uses the operator < for comparisons.
```

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.

**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**
- `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.

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.

**Return** The `make_heap` algorithm returns a `hpx::future<void>` if the execution policy is of type `task_execution_policy` and returns `void` otherwise.

**namespace hpx**

**Functions**

```
template<typename ExPolicy, typename RandIter1, typename RandIter2, typename RandIter3, typename Comp = detail::less>
    hpx::parallel::util::detail::algorithm_result<ExPolicy, RandIter3>::type 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.
```

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.

**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:

```cpp
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`.

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.

**Return** The `merge` algorithm returns a `hpx::future<RandIter3>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `tagged_tuple<RandIter3>` otherwise. The `merge` algorithm returns the destination iterator to the end of the `dest` range.

```
template<typename ExPolicy, typename RandIter, typename Comp = detail::less>
util::detail::algorithm_result<ExPolicy>::type 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.
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.

**Note** Complexity: Performs \( O(\text{std::distance}(\text{first}, \text{last})) \) 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:

```cpp
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`.

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.

**Return** 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`.

```cpp
namespace hpx
```

```cpp
namespace parallel
```
Functions

```cpp
template<typename ExPolicy, typename FwdIter, typename Proj = util::projection_identity, typename F = detail::less>
util::detail::algorithm_result<ExPolicy, FwdIter>::type min_element(ExPolicy&& policy, FwdIter first, FwdIter 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.

**Note** Complexity: Exactly \( \max(N-1, 0) \) comparisons, where \( N = \text{std::distance(first, last)} \).

**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`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Proj = util::projection_identity, typename F = detail::less>
util::detail::algorithm_result<ExPolicy, FwdIter>::type max_element(ExPolicy&& policy, FwdIter first, FwdIter 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.

**Note** Complexity: Exactly \( \max(N-1, 0) \) comparisons, where \( N = \text{std::distance(first, last)} \).

**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`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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.

The comparisons in the parallel `max_element` algorithm invoked with an execution policy object of type `sequenced_task_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Proj = util::projection_identity, typename F = detail::less>
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::min(FwdIter), tag::max
FwdIter>>, type minmax_element ExPolicy &&policy, FwdIter first, FwdIter 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.

**Note** Complexity: At most \( \max(\text{floor}(3/2^*(N-1)), 0) \) applications of the predicate, where \( N = \text{std::distance(first, last)} \).

**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`.

• **Proj**: The type of an optional projection function. This defaults to `util::projection_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:

  ```cpp
  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.

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.

### Return

The `minmax_element` algorithm returns a `hpx::future<tagged_pair<tag::min(FwdIter), tag::max(FwdIter)>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `tagged_pair<tag::min(FwdIter), tag::max(FwdIter)>` otherwise. The `minmax_element` algorithm returns a pair consisting of an iterator to the smallest element as the first element and an iterator to the greatest element as the second. 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.

```
namespace hpx

Functions

template<
typeName ExPolicy, typeName FwdIter1, typeName FwdIter2, typeName Pred = detail::equal_to>
util::detail::algorithm_result_t<
ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch(
ExPolicy &&policy, 
FwdIter1 first1, FwdIter1 last1,
FwdIter2 first2, FwdIter2 last2,
Pred &&op = Pred())
```

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.

**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.

### Template Parameters
• **ExPolicy**: The type of the execution policy to 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.

• **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:

```cpp
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.

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** The two ranges are considered mismatch if, for every iterator `i` in the range `[first1, last1)`, *(first2 + (i - first1)). This overload of mismatch uses operator== to determine if two elements are mismatch.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = detail::equal_to>
util::detail::algorithm_result_t<ExPolicy, std::pair<FwdIter1, FwdIter2>> mismatch (ExPolicy &&policy, FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, Pred &&op = Pred())
```

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.

**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.

- `FwdIter1`: 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 `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:

```cpp
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.

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.

**Return**  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 ExPolicy, typename FwdIter1, typename FwdIter2>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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.

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.

**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.

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.

**Return** The move algorithm returns a hpx::future<tagged_pair<tag::in(FwdIter1), tag::out(FwdIter2)>> if the execution policy is of type sequenced_task_policy or parallel_task_policy and returns tagged_pair<tag::in(FwdIter1), tag::out(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.

namespace hpx
Functions

template<typename RandomIt, typename Pred>
void nth_element (RandomIt first, RandomIt nth, RandomIt last, 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 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.

• 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:

```cpp
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<>.

Return The nth_element algorithms returns nothing.

template<typename ExPolicy, typename RandomIt, typename Pred>
void nth_element (ExPolicy &&policy, RandomIt first, RandomIt nth, RandomIt last, 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.

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:

```cpp
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<>`.

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.

**Return** The `nth_element` algorithms returns nothing.

### namespace hpx

#### Functions

```cpp
template<typename RandIter, typename Comp>
void partial_sort (RandIter first, RandIter middle, RandIter last, Comp &&comp = Comp())
```

Placing 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.

**Return** The `partial_sort` algorithm returns nothing.

#### 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.
• 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.

template<typename ExPolicy, typename RandIter, typename Comp>
util::detail::algorithm_result_t<ExPolicy> 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.

Return The partial_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.

Template Parameters

• ExPolicy: The type of the execution 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.

namespace hpx

Functions

template<typename InIter, typename RandIter, typename Comp>
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.

**Note** Complexity: \( O(N \log(\min(D, N))) \), where \( N = \text{std::distance(first, last)} \) and \( D = \text{std::distance(d_first, d_last)} \) 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 a 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`.

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.

**Return** The `partial_sort_copy` algorithm returns a returns `RandomIt`. The algorithm returns an iterator to the element defining the upper boundary of the sorted range i.e. \( d\text{\_first} + \min(last - first, d\text{\_last} - d\text{\_first}) \)

```cpp
template<
    typename ExPolicy,
    typename FwdIter,
    typename RandIter,
    typename Comp
>
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
)
```

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 = \text{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.

**Note** Complexity: \( O(N \log(\min(D, N))) \), where \( N = \text{std::distance(first, last)} \) and \( D = \text{std::distance(d_first, d_last)} \) 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.

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.

**Return**  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)`

namespace hpx

**Functions**

```cpp
template<
    typename FwdIter,
    typename Pred,
    typename Proj
>
FwdIter partition(ExPolicy &&policy, FwdIter first, FwdIter last, Pred &&pred, 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.
• **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 `util::projection_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:

```cpp
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.

**Return** The `partition` algorithm returns returns `FwdIter`. The `partition` algorithm returns the iterator to the first element of the second group.

```cpp
template<typename ExPolicy, typename FwdIter, typename Pred, typename Proj> util::detail::algorithm_result_t<ExPolicy, FwdIter> partition(ExPolicy &&policy, FwdIter first, FwdIter last, Pred &&pred, 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.

**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 `util::projection_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:

```cpp
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.

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.

Return  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>
BidirIter stable_partition(BidirIter first, BidirIter last, F &&f, 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) \( \text{INVOKE}(f, \text{INVOKE}(\text{proj}, *j)) \neq \text{false} \), and for every iterator k in the range [i, last), \( \text{INVOKE}(f, \text{INVOKE}(\text{proj}, *k)) == \text{false} \)

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 \( (\text{last} - \text{first}) \times \log(\text{last} - \text{first}) \) swaps, but only linear number of swaps if there is enough extra memory. Exactly \( \text{last} - \text{first} \) 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.

• 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 util::projection_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:

```cpp
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.

**Return** The *stable_partition* algorithm returns an iterator \( i \) such that for every iterator \( j \) in the range \([\text{first}, i)\), \( \text{INVOKE}(f, \text{INVOKE}(\text{proj}, *j)) \neq \text{false} \), and for every iterator \( k \) in the range \([i, \text{last})\), \( f(*k) == \text{false} \) \( \text{INVOKE}(f, \text{INVOKE}(\text{proj}, *k)) == \text{false} \). The relative order of the elements in both groups is preserved.

```cpp
template<typename ExPolicy, typename BidirIter, typename F, typename Proj>
util::detail::algorithm_result_t<ExPolicy, BidirIter> stable_partition(ExPolicy &&policy, BidirIter first, BidirIter last, F &&f, Proj &&proj)
```

Permutates the elements in the range \([\text{first}, \text{last})\) such that there exists an iterator \( i \) such that for every iterator \( j \) in the range \([\text{first}, i)\) \( \text{INVOKE}(f, \text{INVOKE}(\text{proj}, *j)) \neq \text{false} \), and for every iterator \( k \) in the range \([i, \text{last})\), \( f(*k) == \text{false} \) \( \text{INVOKE}(f, \text{INVOKE}(\text{proj}, *k)) == \text{false} \).

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.

**Note** Complexity: At most \((\text{last} - \text{first}) \times \log(\text{last} - \text{first})\) swaps, but only linear number of swaps if there is enough extra memory. Exactly \(\text{last} - \text{first}\) 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.
- **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 *util::projection_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 \([\text{first}, \text{last})\). The signature of this predicate should be equivalent to:

```cpp
bool fun(const Type &a);
```

The signature does not need to have \( \text{const} \& \). 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.

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.
Return  The stable_partition algorithm returns an iterator $i$ such that for every iterator $j$ in the range $[\text{first}, i)$, $f(*j) \neq \text{false}$ and for every iterator $k$ in the range $[i, \text{last})$, $f(*k) = \text{false}$. 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.

```cpp
template<typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
    hpx::util::tagged_tuple<
        tag::in(FwdIter1),
        tag::out1 FwdIter2,
        tag::out2 FwdIter3>
    partition_copy
    FwdIter1 first,
    FwdIter1 last,
    FwdIter2 dest_true,
    FwdIter3 dest_false,
    Pred && pred,
    Proj && proj

Copies the elements in the range, defined by $[\text{first}, \text{last})$, 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 $\text{dest_true}$. The rest of the elements are copied to the range beginning at $\text{dest_false}$. 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 \text{last} - \text{first} assignments, exactly \text{last} - \text{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 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 util::projection_identity

Parameters

- $\text{first}$: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- $\text{last}$: Refers to the end of the sequence of elements the algorithm will be applied to.
- $\text{dest_true}$: Refers to the beginning of the destination range for the elements that satisfy the predicate $pred$
- $\text{dest_false}$: 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 $[\text{first}, \text{last})$. This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```cpp
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.

**Return** The `partition_copy` algorithm returns `tagged_tuple<tag::in(InIter), tag::out1(OutIter1), tag::out2(OutIter2)>`. 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_tuple<tag::in(FwdIter1), tag::out1 FwdIter2, tag::out2FwdIter3>,::type partition_copyExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest_true, FwdIter3 dest_false, Pred &&pred, Proj &&projCopies 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.

**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 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 `util::projection_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:

```cpp
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.

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.

**Return** The `partition_copy` algorithm returns a `hpx::future<tagged_tuple<tag::in(InIter), tag::out1(OutIter1), tag::out2(OutIter2)>>` if the execution policy is of type `parallel_task_policy` and returns `tagged_tuple<tag::in(InIter), tag::out1(OutIter1), tag::out2(OutIter2)>>` 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.

```cpp
namespace hpx
{

Functions

```cpp
template<typename ExPolicy, typename FwdIter, typename T, typename F>
util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, FwdIter first, FwdIter 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.

**Note** Complexity: `O(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 begin and end 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 `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.
• \textit{f}: Specifies the function (or function object) which will be invoked for each of the elements in
the sequence specified by \([\text{first, last})\). This is a binary predicate. The signature of this predicate
should be equivalent to:

\[
\text{Ret fun(const Type1 &a, const Type1 &b);}
\]

The signature does not need to have \text{const}\&. The types \text{Type1 Ret}
must be such that an object of
type \text{FwdIter} can be dereferenced and then implicitly converted to any of those types.

• \textit{init}: The initial value for the generalized sum.

The reduce operations in the parallel \textit{copy_if} algorithm invoked with an execution policy object of type
\text{parallel_policy} or \text{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified
threads, and indeterminately sequenced within each thread.

The difference between \textit{reduce} and \textit{accumulate} is that the behavior of reduce may be non-deterministic for
non-associative or non-commutative binary predicate.

\textbf{Return} The \textit{reduce} algorithm returns a \text{hpx::future<T>} if the execution policy is of type \text{sequenced_task_policy} or \text{parallel_task_policy}
and returns \text{T} otherwise. The \textit{reduce} algorithm returns
the result of the generalized sum over the elements given by the input range \([\text{first, last})\).

\textbf{Note} \text{GENERALIZED_SUM(op, a1, \ldots, aN)} is defined as follows:

\begin{itemize}
  \item \(a1\) when \(N\) is 1
  \item \(\text{op}(\text{GENERALIZED_SUM(op, b1, \ldots, bK), GENERALIZED_SUM(op, bM, \ldots, bN)})\), where:
    \begin{itemize}
      \item \(b1, \ldots, bN\) may be any permutation of \(a1, \ldots, aN\) and
      \item \(1 < K+1 = M \leq N\).
    \end{itemize}
\end{itemize}

\[
\text{template<typename ExPolicy, typename FwdIter, typename T>}
\text{util::detail::algorithm_result<ExPolicy, T>::type reduce (ExPolicy &&policy, FwdIter first, FwdIter}
\text{last, T init)}
\]

Returns \text{GENERALIZED_SUM(+, init, *first, \ldots, *(first + (last - first) - 1))}.

The reduce operations in the parallel \textit{reduce} algorithm invoked with an execution policy object of type
\text{sequenced_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: \(O(\text{last} - \text{first})\) applications of the operator+(1).

\textbf{Template Parameters}

\begin{itemize}
  \item \textit{ExPolicy}: The type of the execution policy to use (deduced). It describes the manner in which
  the execution of the algorithm may be parallelized and the manner in which it executes the as-
  signments.
  \item \textit{FwdIter}: The type of the source begin and end iterators used (deduced). This iterator type must
  meet the requirements of an forward iterator.
  \item \textit{T}: The type of the value to be used as initial (and intermediate) values (deduced).
\end{itemize}

\textbf{Parameters}

\begin{itemize}
  \item \textit{policy}: The execution policy to use for the scheduling of the iterations.
  \item \textit{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
  \item \textit{last}: Refers to the end of the sequence of elements the algorithm will be applied to.
  \item \textit{init}: The initial value for the generalized sum.
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.

Return 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).

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<typename ExPolicy, typename FwdIter>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIter>::value_type>::type
reduce(ExPolicy &&policy, FwdIter first, FwdIter last)

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.

Note Complexity: O(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 begin and end 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.

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.

Return 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).

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.

namespace hpx

namespace parallel

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.

comp has to induce a strict weak ordering on the values.

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 an forward iterator.
- **FwdIter2**: The type of the iterator representing the destination value range (deduced). This iterator type must meet the requirements of an 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 \( copy_if \) requires \( F \) 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:

```cpp
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.

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.

Return 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> oth-
namespace hpx

Functions

template<typename FwdIter, typename T>
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

• ExPolicy: The type of the execution policy to 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.

Return 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>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.

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.

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.

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.

Return

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.

```cpp
template<typename FwdIter, typename Pred>
FwdIter remove(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:

```cpp
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`.

Return

The `remove_if` algorithm returns a `FwdIter`. The `remove_if` algorithm returns the iterator to the new end of the range.

```cpp
template<typename ExPolicy, typename FwdIter, typename Pred>
```
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.

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.

**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 a 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:

```cpp
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`.

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.

**Return** 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.

**namespace hpx**
Functions

template<typename InIter, typename OutIter, typename T>
FwdIter 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 f.

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

- 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.
- val: Value to be removed.

Return 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>
FwdIter 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.

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.

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.

• 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.
• val: Value to be removed.

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.

Return 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.

\[
\text{template<typename InIter, typename OutIter, typename Pred>}
\]

FwdIter 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 \( f \) 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 \( f \).

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).

• 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:

```cpp
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`.

**Return** 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>
FwdIter 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 `f` returns false. The order of the elements that are not removed is preserved.

**Effects**: Copies all the elements referred to by the iterator in the range [first,last) for which the following corresponding conditions do not hold: \[\text{INVOKE}(\text{pred}, \*\text{it}) \neq \text{false}\].

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.

**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 `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.

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.

Return 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.

namespace hpx

Functions

template<typename InIter, typename T>
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.

Return The replace algorithm returns a void.

template<typename ExPolicy, typename FwdIter, typename T>
parallel::util::detail::algorithm_result<ExPolicy, void>::type 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).

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.

**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.

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.

**Return** 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>
void replace_if(Iter first, Iter 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).`

**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:

```cpp
template<typename ExPolicy, typename FwdIter, typename Pred, typename T>
parallel::util::detail::algorithm_result<ExPolicy, void>::type 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) 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: \( \text{INVOLVE}(f, *\text{it}) \neq \text{false} \)

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.

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:

```cpp
bool pred(const Type &a);
```
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.

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.

Return 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>
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. of the algorithm may be parallelized and the manner in which it executes the assignments.

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.

Return 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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.

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.

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.

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.

Return  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.
OutIter replace_copy_if(InIter first, InIter last, OutIter dest, Pred &&pred, T const &new_value)

Copies the all elements from the range \([\text{first}, \text{last})\) to another range beginning at \(\text{dest}\) replacing all elements satisfying a specific criteria with \(\text{new\_value}\).

Effects: Assigns to every iterator \(\text{it}\) in the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) either \(\text{new\_value}\) or \(*((\text{first} + (\text{it} - \text{result}))))\) depending on whether the following corresponding condition holds: INVOKE(f, *((\text{first} + (\text{it} - \text{result})))) != false

The assignments in the parallel replace_copy_if algorithm execute in sequential order in the calling thread.

**Note** Complexity: Performs exactly \(\text{last} - \text{first}\) 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 \(\text{Pred}\) 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 \([\text{first}, \text{last})\). This is an unary predicate which returns true for the elements which need to replaced. The signature of this predicate should be equivalent to:

```c++
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.

**Return** 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.
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.

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 equal requires F 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:

```cpp
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.

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.

Return 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.
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 an 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.

Return The reverse algorithm returns a void.

template<typename ExPolicy, typename BidirIter>
parallel::util::detail::algorithm_result<ExPolicy, void>::type 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.

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.

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 an 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.

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.

Return 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_first in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(dest_first + (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_first, dest_first+(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 an 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.

Return 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>

util::detail::algorithm_result<ExPolicy, FwdIter>::type reverse_copy(ExPolicy &&policy, BidirIter first, BidirIter last, FwdIter dest_first)

Copies the elements from the range [first, last) to another range beginning at dest_first in such a way that the elements in the new range are in reverse order. Behaves as if by executing the assignment *(dest_first + (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_first, dest_first+(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.

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 an bidirectional 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.
• **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.

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.

**Return** 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.

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.

**Note** Complexity: Linear in the distance between `first` and `last`.

**Note** The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

**Return** 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 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.
• **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. The assignments in the parallel `rotate` algorithm execute in sequential order in the calling thread.

```
template<
typename ExPolicy, typename FwdIter>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.

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.


**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 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.
- **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.

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**  The type of dereferenced FwdIter must meet the requirements of MoveAssignable and MoveConstructible.

**Return**  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).

```cpp
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 an forward iterator.
- **OutIter**: The type of the source iterators used (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.
- **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.
Return 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>
util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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.

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

- `ExPolicy`: The type of the execution policy to 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.

- `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.

The assignments in the parallel `rotate_copy` algorithm execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return 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.

namespace hpx
Functions

template<typename FwdIter, typename FwdIter2, typename Pred = 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 $S = \text{distance}(s\text{-}first, s\text{-}last)$ and $N = \text{distance}(\text{first, last})$.

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.
- 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 adjacent_find 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: 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

```cpp
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.

Return 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 = detail::equal_to>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.
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. 

**Note** Complexity: at most \((S*N)\) comparisons where \(S = \text{distance}(s\_first, s\_last)\) and \(N = \text{distance}(\text{first}, \text{last})\).

**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.
- **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 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 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

\[
\text{bool pred(const Type1 &a, const Type2 &b);}\]

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.

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.

**Return** 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 \([\text{first}, \text{last})\). If the length of the subsequence \([s\_first, s\_last)\) is greater than the length of the range \([\text{first}, \text{last})\), 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 = 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 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 adjacent_find 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

```cpp
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

Return 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 = detail::equal_to> util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.
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.

**Note** Complexity: at most (S*N) comparisons where \( S = \text{distance(s\_first, s\_last)} \) and \( N = \text{count} \).

### 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.
- **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 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

  ```cpp
  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.

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.

**Return** 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 \([first, first+\text{count})\). If the length of the subsequence \([s\_first, s\_last)\) is greater than the length of the range \([first, first+\text{count})\), `first` is returned. Additionally if the size of the subsequence is empty or no subsequence is found, `first` is also returned.
Functions

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = detail::less>
util::detail::algorithm_result<ExPolicy, FwdIter3>::type::type
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 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.

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.

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).

Template Parameters

- **ExPolicy**: The type of the execution 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 an forward iterator.

- **FwdIter2**: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.

- **FwdIter3**: 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<>

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:

```cpp
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`.

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.

Return 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.

```cpp
namespace hpx
```

### Functions

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = detail::less>
util::detail::algorithm_result<ExPolicy, FwdIter3>::type 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 `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.

**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.

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`).

### Template Parameters
• **ExPolicy**: The type of the execution 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 an forward iterator.

• **FwdIter2**: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.

• **FwdIter3**: 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<>`

### 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:

```cpp
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`.

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.

### Return

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.
Functions

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename Pred = detail::less>
util::detail::algorithm_result<ExPolicy, FwdIter3>::type::type
set_symmetric_difference(
  ExPolicy &&policy,
  FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2, FwdIter3 dest, Pred &&op
)

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.

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.

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).

Template Parameters

- ExPolicy: The type of the execution 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 an forward iterator.

- FwdIter2: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.

- FwdIter3: 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 `Copy-Constructible`. 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:

  ```cpp
  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`.

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.

**Return** 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.

```cpp
namespace hpx

Functions

template<
  typename ExPolicy,
  typename FwdIter1,
  typename FwdIter2,
  typename FwdIter3,
  typename Pred = detail::less>
util::detail::algorithm_result<ExPolicy, FwdIter3>::type
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 `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.
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.

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).

Template Parameters

- ExPolicy: The type of the execution 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 an forward iterator.
- FwdIter2: The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- FwdIter3: 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 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:

```cpp
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.

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.
Return 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.

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.

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 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.

Note The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignble`.

Return 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>
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`.

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.

Note Complexity: At most `(last - first) - n` 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 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.
- **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.

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** The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

**Return** 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.

```cpp
namespace hpx {

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.

**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 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.
Note The type of dereferenced FwdIter must meet the requirements of MoveAssignable.

Return 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>
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.

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.

Note Complexity: At most (last - first) - n 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 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.
- 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.

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 The type of dereferenced FwdIter must meet the requirements of MoveAssignable.

Return 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.

namespace hpx
Functions

template<typename RandomIt>
void sort (RandomIt first, RandomIt last)
    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.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

The assignments in the parallel sort algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Template Parameters
    • RandomIt: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access 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.

Return The sort algorithm does not return anything.

template<typename ExPolicy, typename RandomIt>
util::detail::algorithm_result<ExPolicy>::type sort (ExPolicy &&policy, RandomIt first, RandomIt last)
    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.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

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.

Template Parameters
    • ExPolicy: The type of the execution 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.
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.

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.

**Return** The `sort` algorithm returns a `hpx::future<void>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns nothing otherwise.

```cpp
template<typename RandomIt, typename Comp, typename Proj>
void sort(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<()`).

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`.

**Note** Complexity: O(Nlog(N)), where N = `std::distance(first, last)` comparisons.

`comp` has to induce a strict weak ordering on the values.

**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 `util::projection_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.

The assignments in the parallel `sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

**Return** The `sort` algorithm returns nothing.

```cpp
template<typename ExPolicy, typename RandomIt, typename Comp, typename Proj>
```
**parallel::util::detail::algorithm_result<ExPolicy>::type 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<()).

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.

**Note** Complexity: \(O(N\log(N))\), where \(N = \text{std::distance}(\text{first}, \text{last})\) comparisons.

*comp* has to induce a strict weak ordering on the values.

**Template Parameters**

- **ExPolicy**: The type of the execution 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 *util::projection_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.

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.

**Return** The *sort* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

```cpp
namespace hpx

namespace parallel
```
Functions

template<typename ExPolicy, typename KeyIter, typename ValueIter, typename Compare = detail::less>
util::detail::algorithm_result<ExPolicy, hpx::util::tagged_pair<tag::in1 (KeyIter), tag::in2 ValueIter>>::type 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<()).

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.

Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons. comp has to induce a strict weak ordering on the values.

Template Parameters

- **ExPolicy**: The type of the execution 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.
- **Comp**: 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.

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.

Return The sort_by-key algorithm returns a hpx::future<tagged_pair<tag::in1(KeyIter), tag::in2(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.

namespace hpx
Functions

template<typename RandomIt>
void stable_sort (RandomIt first, RandomIt last)
    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.

    Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

The assignments in the parallel stable_sort algorithm invoked without an execution policy object execute
in sequential order in the calling thread.

Template Parameters

• RandomIt: The type of the source iterators used (deduced). This iterator type must meet the
  requirements of a random access 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.

Return The stable_sort algorithm does not return anything.

template<typename ExPolicy, typename RandomIt>
util::detail::algorithm_result<ExPolicy>::type stable_sort (ExPolicy &&policy, RandomIt first, RandomIt last)
    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.

    Note Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

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.

Template Parameters

• ExPolicy: The type of the execution 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.

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.

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.

**Return** 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 nothing otherwise.

```
template<typename RandomIt, typename Comp, typename Proj>
void stable_sort(RandomIt first, RandomIt last, Comp &comp, 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`.

**Note** Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons.

`comp` has to induce a strict weak ordering on the values.

**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 `util::projection_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.

The assignments in the parallel `stable_sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

**Return** The `stable_sort` algorithm returns nothing.

```
template<typename ExPolicy, typename RandomIt, typename Comp, typename 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`.

**Note** Complexity: $O(N \log(N))$, where $N = \text{std::distance(first, last)}$ comparisons.

`comp` has to induce a strict weak ordering on the values.

**Template Parameters**

- `ExPolicy`: The type of the execution 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 `util::projection_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.

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.

**Return** 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 nothing otherwise.
Variables

\textit{hpx::starts\_with\_t} \textbf{starts\_with}

\textbf{struct starts\_with\_t} : \textbf{public hpx::functional::detail::tag\_fallback<starts\_with\_t>}

Friends

\begin{verbatim}
template<typename InIter1, typename InIter2, typename Pred = hpx::parallel::v1::detail::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
bool tag\_fallback\_invoke (hpx::starts\_with\_t, InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred = ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
parallel::util::detail::algorithm\_result<ExPolicy, bool>::type tag\_fallback\_invoke (hpx::starts\_with\_t, ExPolicy &&policy, FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2, Pred &&pred = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())
\end{verbatim}

namespace hpx

Functions

\begin{verbatim}
template<typename FwdIter1, typename FwdIter2>
FwdIter2 swap\_ranges (FwdIter1 first1, FwdIter1 last1, FwdIter2 first2)
    Exchanges elements between range \texttt{[first1, last1]} and another range starting at \texttt{first2}.
\end{verbatim}

The swap operations in the parallel \texttt{swap\_ranges} algorithm invoked without an execution policy object execute in sequential order in the calling thread.

\textbf{Note} Complexity: Linear in the distance between \texttt{first1} and \texttt{last1}

Template Parameters

- \texttt{FwdIter1}: The type of the first range of iterators to swap (deduced). This iterator type must meet the requirements of an forward iterator.
• FwdIter2: The type of the second range of iterators to swap (deduced). This iterator type must meet the requirements of an forward iterator.

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.

Return 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type swap_ranges(ExPolicy &&policy,
    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 with an execution policy object of type sequenced_policy execute in sequential order in the calling 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 an forward iterator.
• FwdIter2: The type of the second range of iterators to swap (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 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.

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.

Return 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.
namespace hpx

Functions

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename F>
util::detail::algorithm_result<ExPolicy, util::in_out_result<FwdIter1, FwdIter2>>::type 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.

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.

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 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.
- 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:

```cpp
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.
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.

**Return** The transform algorithm returns a `hpx::future<in_out_result<FwdIter1, FwdIter2>>` if the execution policy is of type parallel_task_policy and returns `in_out_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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename FwdIter3, typename F>
util::detail::algorithm_result<ExPolicy, util::in_in_out_result<FwdIter1, FwdIter2, FwdIter3>>::type transform(ExPolicy &&policy, FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 first2, FwdIter3 dest, F &&f)
```

Applies the given function \( f \) to pairs of elements from two ranges: one defined by \([\text{first1}, \text{last1})\) and the other beginning at \( \text{first2} \), and stores the result in another range, beginning at \( \text{dest} \).

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.

**Note** Complexity: Exactly \( \text{last} - \text{first} \) 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 an forward iterator.
- **FwdIter2**: The type of the source iterators for the second range used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter3**: The type of the iterator representing the destination range (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 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:

```cpp
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.

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.

**Return**  The transform algorithm returns a hpx::future<in_in_out_result<FwdIter1, FwdIter2, FwdIter3>> if the execution policy is of type parallel_task_policy and returns in_in_out_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.

**namespace hpx**

**Functions**

```
template<
        typename InIter,
        typename OutIter,
        typename T,
        typename BinOp,
        typename UnOp>

OutIter transform_exclusive_scan(InIter first, 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.

**Note**  Complexity: O(last - first) applications of the predicates op and conv.

**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.
• **Conv**: 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).
• **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.
• **conv**: 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:

```cpp
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.
• **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:

```cpp
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.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

**Return** 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.

**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`.

The behavior of `transform_exclusive_scan` may be non-deterministic for a non-associative predicate.
Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \( \text{GENERALIZED_NONCOMMUTATIVE\_SUM}(\text{binary\_op}, \text{init}, \text{conv}(*\text{first}), \ldots, \text{conv}(*(\text{first} + (i - \text{result}) - 1))) \).

The reduce operations in the parallel \( \text{transform\_exclusive\_scan} \) algorithm invoked with an execution policy object of type \( \text{sequenced\_policy} \) execute in sequential order in the calling thread.

**Note** Complexity: \( O(\text{last} - \text{first}) \) applications of the predicates \( \text{op} \) and \( \text{conv} \).

**Template Parameters**

- **ExPolicy**: The type of the execution policy to 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.

- **Conv**: 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).

- **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.

- **conv**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a unary predicate. The signature of this predicate should be equivalent to:

```cpp
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 \( \text{Type} \) must be such that an object of type \( \text{FwdIter1} \) 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.
- **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:

```cpp
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.

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 subranges, or modify elements in the ranges \([\text{first}, \text{last})\) or \([\text{result} + (\text{last} - \text{first}), \text{result} + (\text{last} - \text{first}))\).

**Return** 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.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN)` is defined as:

- \( a1 \) when \( N = 1 \)
- \( \text{op}(\text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, a1, ..., aK), \text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, aM, ..., aN)) \) where \( 1 < K+1 = M \leq N \).

The behavior of `transform_exclusive_scan` may be non-deterministic for a non-associative predicate.

```cpp
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 \([\text{result}, \text{result} + (\text{last} - \text{first}))\) 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.

**Note** Complexity: \( O(\text{last} - \text{first}) \) applications of the predicate `op`.

**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.
- **Conv**: The type of the unary function object used for the conversion operation.
- **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:

  ```cpp
  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.

- **conv**: 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:

  ```cpp
  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`.

Neither **conv** nor **op** shall invalidate iterators or subranges, or modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

**Return** 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.

**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`.

The difference between `inclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the `ith` input element in the `ith` sum.
Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \( \text{GENERALIZED_NONCOMMUTATIVE}\_\text{SUM}(\text{op}, \text{conv}(*\text{first}), \ldots, \text{conv}(*(\text{first} + (i - \text{result})))). \)

The reduce operations in the parallel \texttt{transform\_inclusive\_scan} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

**Note** Complexity: \( O(\text{last} - \text{first}) \) applications of the predicate \( \text{op} \).

**Template Parameters**

- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{FwdIter1}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{FwdIter2}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{Conv}: The type of the unary function object used for the conversion operation.
- \texttt{Op}: The type of the binary function object used for the reduction operation.

**Parameters**

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \texttt{last}: Refers to the end of the sequence of elements the algorithm will be applied to.
- \texttt{dest}: Refers to the beginning of the destination range.
- \texttt{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:

  ```cpp
  Ret\ fun(const\ Type1 \&a, const\ Type1 \&b);
  ```

  The signature does not need to have \texttt{const\&}, but the function must not modify the objects passed to it. The types \texttt{Type1} and \texttt{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.
• \texttt{conv}: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a unary predicate. The signature of this predicate should be equivalent to:

\begin{verbatim}
R fun(const Type &a);
\end{verbatim}

The signature does not need to have const&, but the function must not modify the objects passed to it. The type \textit{Type} must be such that an object of type \textit{FwdIter1} can be dereferenced and then implicitly converted to \textit{Type}. The type \textit{R} must be such that an object of this type can be implicitly converted to \textit{T}.

The reduce operations in the parallel \textit{transform_inclusive_scan} algorithm invoked with an execution policy object of type \textit{parallel_policy} or \textit{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither \textit{conv} nor \textit{op} shall invalidate iterators or subranges, or modify elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

\textbf{Return} The \textit{transform_inclusive_scan} algorithm returns a \texttt{hpx::future<FwdIter2>} if the execution policy is of type \textit{sequenced_task_policy} or \textit{parallel_task_policy} and returns \texttt{FwdIter2} otherwise. The \textit{transform_inclusive_scan} algorithm returns the output iterator to the element in the destination range, one past the last element copied.

\textbf{Note} \texttt{GENERALIZED_NONCOMMUTATIVE\_SUM}(\textit{op}, \texttt{a1}, \ldots, \texttt{aN}) is defined as:

\begin{itemize}
  \item \texttt{a1} when \textit{N} is 1
  \item \texttt{op(\texttt{GENERALIZED_NONCOMMUTATIVE\_SUM}(\textit{op}, \texttt{a1}, \ldots, \texttt{aK}), \texttt{GENERALIZED_NONCOMMUTATIVE\_SUM}(\textit{op}, \texttt{aM}, \ldots, \texttt{aN}))} where \(1 < \texttt{K+1 = M} \leq \texttt{N}\).
\end{itemize}

The difference between \textit{inclusive_scan} and \textit{transform_inclusive_scan} is that \textit{transform_inclusive_scan} includes the \textit{i}th input element in the \textit{i}th sum.

\begin{verbatim}
template<typename InIter, typename OutIter, typename BinOp, typename UnOp, typename T>
OutIter transform_inclusive_scan(InIter first, InIter last, OutIter dest, BinOp &&binary_op,
UnOp &&unary_op, T init)
\end{verbatim}

Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\textit{op}, \texttt{init}, \texttt{conv(*first)}, \ldots, \texttt{conv(*((first + (i - result)))).}

The reduce operations in the parallel \textit{transform_inclusive_scan} algorithm invoked without an execution policy object execute in sequential order in the calling thread.

\textbf{Note} Complexity: \(O(\text{last} - \text{first})\) applications of the predicate \textit{op}.

\textbf{Template Parameters}

\begin{itemize}
  \item \texttt{InIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
  \item \texttt{OutIter}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
  \item \texttt{Conv}: The type of the unary function object used for the conversion operation.
  \item \texttt{Op}: The type of the binary function object used for the reduction operation.
  \item \texttt{T}: The type of the value to be used as initial (and intermediate) values (deduced).
\end{itemize}

\textbf{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:

```cpp
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.
• **conv**: 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:

```cpp
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.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges `[first, last)` or `[result, result + (last - first))`.

**Return** 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.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aN)` is defined as:

• a1 when N is 1
• `op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aK), GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, \ldots, aN))` where 1 < K+1 = M <= N.

The difference between `inclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the ith input element in the ith sum. `op` is not mathematically associative, the behavior of `transform_inclusive_scan` may be non-deterministic.
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)))).

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.

**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 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.
- Conv: The type of the unary function object used for the conversion operation.
- 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:

```cpp
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.

- **conv**: 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:

```cpp
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.

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 subranges, or modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

**Return** 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.

**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`.

The difference between `inclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the `i`th input element in the `i`th sum. `op` is not mathematically associative, the behavior of `transform_inclusive_scan` may be non-deterministic.

```
namespace hpx
{

Functions

```cpp

 template<typename ExPolicy, typename FwdIter, typename T, typename Reduce, typename Convert>
 util::detail::algorithm_result<ExPolicy, T>::type 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))))`.

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.

**Note** Complexity: `O(last - first)` applications of the predicates `red_op` and `conv_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 executes the assignments.

• **FwdIter**: 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 `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).

• **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.

• **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:

```cpp
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`.

• **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:

```cpp
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.

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.

**Return** 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)`.

**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<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename T>
util::detail::algorithm_result<ExPolicy, T>::type 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.

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.

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.
• FwdIter1: The type of the first source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
• FwdIter2: 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.
• 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.

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.

Return 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.
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.

**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.
- **FwdIter1**: The type of the first source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- **FwdIter2**: 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.
- **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 $op2$. This is a binary predicate. The signature of this predicate should be equivalent to:

```cpp
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:

```cpp
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 $op1$. 
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.

**Return** 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.

```cpp
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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type
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.

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.
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.

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.

Return 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.

```cpp
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
- **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.

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.
**Return** The `uninitialized_copy_n` algorithm returns a return `FwdIter2`. 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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.

**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 input iterator.
- `Size`: The type of the argument specifying the number of elements to apply 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.

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.

**Return** 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.
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 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.

Return The uninitialized_default_construct algorithm returns nothing

template<typename ExPolicy, typename FwdIter>
parallel::util::detail::algorithm_result<ExPolicy>::type 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.

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.

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 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.

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.

**Return** 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.

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 an 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.

**Return** 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.

**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.

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.

### Return

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.

```cpp
namespace hpx
{
    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.
    }
}
```

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

- **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.

### Return

The *uninitialized_fill* algorithm returns nothing.
**parallel::util::detail::algorithm_result<ExPolicy>::type 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.

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.

**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 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.

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.

**Return** 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.

The assignments in the parallel uninitialized_fill_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.
- **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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Size, typename T>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type 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 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.

**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.

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.

**Return** 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.

```cpp
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.

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.
- **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.

**Return** 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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.

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.

**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.

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.

**Return** 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>
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.

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**

• **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.

**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.

**Return** The *uninitialized_move_n* algorithm returns a returns `FwdIter2`. 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>
```
Moves the elements in the range \([\text{first}, \text{first} + \text{count})\), starting from \text{first} and proceeding to \(\text{first} + \text{count} - 1\), to another range beginning at \text{dest}. If an exception is thrown during the initialization, some objects in \([\text{first}, \text{first} + \text{count})\) are left in a valid but unspecified state.

The assignments in the parallel \texttt{uninitialized\_move\_n} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Performs exactly \texttt{count} movements, if \texttt{count} > 0, no move operations otherwise.

\textbf{Template Parameters}

- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{FwdIter1}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- \texttt{Size}: The type of the argument specifying the number of elements to apply \texttt{f} to.
- \texttt{FwdIter2}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

\textbf{Parameters}

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \texttt{count}: Refers to the number of elements starting at \texttt{first} the algorithm will be applied to.
- \texttt{dest}: Refers to the beginning of the destination range.

The assignments in the parallel \texttt{uninitialized\_move\_n} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \texttt{uninitialized\_move\_n} algorithm returns a \texttt{hpx::future\<FwdIter2\>} if the execution policy is of type \texttt{sequenced\_task\_policy} or \texttt{parallel\_task\_policy} and returns \texttt{FwdIter2} otherwise. The \texttt{unin}\texttt{itialized\_move\_n} algorithm returns the output iterator to the element in the destination range, one past the last element moved.

\begin{verbatim}namespace hpx { namespace util { namespace detail { namespace algorithm_result<ExPolicy, FwdIter2>::type uninitialized_move_n (ExPolicy &&policy, FwdIter1 first, Size count, FwdIter2 dest) { // Implementation... } } } } 
\end{verbatim}
Functions

template<typename FwdIter>
void uninitialized_value_construct (FwdIter first, FwdIter last)

Constructs objects of type typename iterator_traits<ForwardIter>::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 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.

Return The uninitialized_value_construct algorithm returns nothing

template<typename ExPolicy, typename FwdIter>
parallel::util::detail::algorithm_result<ExPolicy>::type uninitialized_value_construct (ExPolicy &&policy, FwdIter first, FwdIter last)

Constructs objects of type typename iterator_traits<ForwardIter>::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.

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 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.

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.

**Return** 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.

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 an 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.

**Return** 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type 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 `sequence_policy` execute in sequential order in the calling 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.

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.

### Return

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.

```cpp
namespace hpx
{

    // Functions

    template<typename FwdIter>
    FwdIter unique(FwdIter first, FwdIter last)
    {
        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.
    }

    // Template Parameters

    template<typename ExPolicy, typename FwdIter>
    {
        // Parameter

        template<typename ExPolicy, typename FwdIter>
    }
```
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.

**Note** Complexity: Performs not more than `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 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.

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.

**Return** 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.

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 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 `util::projection_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 \([\text{first}, \text{last})\). This is an binary predicate which returns \textit{true} for the required elements. The signature of this predicate should be equivalent to:

```cpp
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have \texttt{const}&, but the function must not modify the objects passed to it. The types \textit{Type1} and \textit{Type2} must be such that objects of types \textit{FwdIter} can be dereferenced and then implicitly converted to both \textit{Type1} and \textit{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.

**Return** The \textit{unique} algorithm returns \textit{FwdIter}. The \textit{unique} algorithm returns the iterator to the new end of the range.

```cpp
template<typename ExPolicy, typename FwdIter, typename Pred, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type unique(ExPolicy &&policy, FwdIter first, FwdIter last, Pred &&pred, Proj &&proj)
```

Eliminates all but the first element from every consecutive group of equivalent elements from the range \([\text{first}, \text{last})\) and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel \textit{unique} algorithm invoked with an execution policy object of type \textit{sequenced_policy} execute in sequential order in the calling thread.

**Note** Complexity: Performs not more than \textit{last} - \textit{first} assignments, exactly \textit{last} - \textit{first} - 1 applications of the predicate \textit{pred} and no more than twice as many applications of the projection \textit{proj}.

**Template Parameters**

• \textit{ExPolicy}: The type of the execution policy to 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.

• \textit{FwdIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

• \textit{Pred}: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \textit{unique} requires \textit{Pred} to meet the requirements of \textit{CopyConstructible}. This defaults to \texttt{std::equal_to}.

• \textit{Proj}: The type of an optional projection function. This defaults to \texttt{util::projection_identity}.

**Parameters**

• \textit{policy}: The execution policy to use for the scheduling of the iterations.
• \textit{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
• \textit{last}: Refers to the end of the sequence of elements the algorithm will be applied to.
• \textit{pred}: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is an binary predicate which returns \textit{true} for the required elements. The signature of this predicate should be equivalent to:
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.

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.

**Return** 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.

```cpp
template<typename InIter, typename OutIter>
OutIter unique_copy(InIter first, InIter last, OutIter dest)
```

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.

**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.

**Return** The `unique_copy` algorithm returns a returns `OutIter`. The `unique_copy` algorithm returns the destination iterator to the end of the `dest` range.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type unique_copy(ExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest)
```

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.

**Note** Complexity: Performs not more than 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.
- **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 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.

**Return** 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.

```cpp
template<typename InIter, typename OutIter, typename Pred, typename Proj>
OutIter unique_copy(InIter first, InIter last, OutIter dest, Pred &&pred, 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.
- **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 util::projection_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:

  ```cpp
  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.

Return The `unique_copy` algorithm returns a returns `OutIter`. The `unique_copy` algorithm returns the destination iterator to the end of the `dest` range.

```cpp
template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Pred, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type unique_copy(ExPolicy &&policy, FwdIter1 first, FwdIter1 last, FwdIter2 dest, Pred &&pred, 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.

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 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 `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 `util::projection_identity`
• **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:

```cpp
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.

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.

**Return**
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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename FwdIter, typename Sent, typename Proj = hpx::parallel::util::projection_identity, typename Pred = detail::equal_to>
FwdIter adjacent_difference(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

**Return**
The adjacent_find algorithm returns an iterator to the first of the identical elements. If no such elements are found, *last* is 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 `util::projection_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:

```cpp
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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Sent, typename Proj = hpx::parallel::util::projection_identity, typename Pred = detail::equal_to>
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.

**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 `util::projection_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:

```cpp
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.
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`.

**Return** 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::parallel::util::projection_identity, typename Pred = detail::equal_to>
    hpx::traits::range_traits<Rng>::iterator_type adjacent_difference(ExPolicy &&policy,
        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

**Return** The `adjacent_difference` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is returned.

**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.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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.

```
template<typename ExPolicy, typename Rng, typename Proj = hpx::parallel::util::projection_identity, typename Pred = detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type
    adjac...
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.

**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). Thhpx::traits::is_range<Rng>::valuee 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 `util::projection_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:

```cpp
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.

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`.

**Return** 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.

Searches the range rng for two consecutive identical elements.
- **Proj**: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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.

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`.

**Return** 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.

```cpp
namespace hpx
namespace ranges

Functions

``template<typename FwdIter, typename Sent, typename Proj = hpx::parallel::util::projection_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

**Return** The `adjacent_find` algorithm returns an iterator to the first of the identical elements. If no such elements are found, `last` is 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 `util::projection_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 \textit{true} if the elements should be treated as equal. The signature should be equivalent to the following:

\[
\text{bool \ pred(\ const \ Type1 \ &a, \ const \ Type1 \ &b);}\]

The signature does not need to have \texttt{const \&}, but the function must not modify the objects passed to it. The types \texttt{Type1} must be such that objects of type \texttt{FwdIter} can be dereferenced and then implicitly converted to \texttt{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 \textit{is} invoked.

\[
\text{template<typename Rng, typename Proj = hpx::parallel::util::projection_identity, typename Pred = detail::equal_to>
\text{hpx::traits::range_traits<Rng>::iterator_type adjacent_find(ExPolicy &&policy, Rng &&rng,}
\quad \text{Pred &&pred = Pred(), Proj &&proj = Proj());}
\]

Searches the range \texttt{rng} for two consecutive identical elements.

**Note** Complexity: Exactly the smaller of \(\text{result - std::begin(rng)} + 1\) and \(\text{std::begin(rng) - std::end(rng)} - 1\) applications of the predicate where \texttt{result} is the value returned

**Return** The \texttt{adjacent\_find} algorithm returns an iterator to the first of the identical elements. If no such elements are found, \texttt{last} is returned.

**Template Parameters**

• \texttt{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
• \texttt{Proj}: The type of an optional projection function. This defaults to \texttt{util::projection\_identity}
• \texttt{Pred}: The type of an optional function/function object to use.

**Parameters**

• \texttt{rng}: Refers to the sequence of elements the algorithm will be applied to.
• \texttt{pred}: The binary predicate which returns \textit{true} if the elements should be treated as equal. The signature should be equivalent to the following:

\[
\text{bool \ pred(\ const \ Type1 \ &a, \ const \ Type1 \ &b);}\]

The signature does not need to have \texttt{const \&}, but the function must not modify the objects passed to it. The types \texttt{Type1} must be such that objects of type \texttt{FwdIter} can be dereferenced and then implicitly converted to \texttt{Type1}.

• \texttt{proj}: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate \textit{is} invoked.

\[
\text{namespace hpx}
\]

\[
\text{namespace ranges}
\]
Functions

template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type 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.

**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 util::projection_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.

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.

**Return** 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.

**Note** Complexity: At most std::distance(begin(rng), end(rng)) applications of the predicate f

**Return** 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.

**Return** 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.

**Return** 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.
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 util::projection_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:

```cpp
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.

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.

Return 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.

```cpp
template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type 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.

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 util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
namespace hpx
namespace ranges

Functions

template<typename ExPolicy, typename Iter1, typename Sent1, typename FwdIter>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_result<Iter1, Iter>>::type copy(ExPolicy &&policy, Iter1 iter, Sent1 sent, FwdIter dest)
```

Copies the elements in the range, defined by [first, last), 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.

**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.
- Iter1: 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.
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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename FwdIter1>
hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_result<typename hpx::traits::range_traits<Rng>::iterator_type, FwdIter1>> copy(ExPolicy&& policy, Rng&& rng, FwdIter1 dest)
```

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.

**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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename Size, typename FwdIter2>
```


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.

**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 \( \text{first} \) the algorithm will be applied to.
- dest: Refers to the beginning of the destination range.

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.

**Return**  The copy_n algorithm returns a \( \text{hpx::future<ranges::copy_n_result<FwdIter1, FwdIter2>}> \) 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.
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.

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.

**Note** Complexity: Performs not more than std::distance(begin(rng), end(rng)) assignments, exactly 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 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.
- **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.
- **Proj**: The type of an optional projection function. This defaults to util::projection_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.
- **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:

```cpp
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.

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.

**Return** 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 OutIter, typename F, typename Proj = hpx::parallel::util::projection_identity>
    hpx::parallel::util::detail::algorithm_result<ExPolicy, hpx::ranges::copy_if_result<typename hpx::traits::range_traits<Rng>::iterator_type, OutIter>>::type
    copy_if(ExPolicy&& policy, Rng&& rng, OutIter dest, F&& f, Proj&& proj = Proj())
```

Copies the elements in the range `rng` 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.

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.

**Note** Complexity: Performs not more than `std::distance(begin(rng), end(rng))` assignments, exactly `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 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.
- **OutIter**: 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 `copy_if` requires `F` to meet the requirements of `CopyConstructible`.
- **Proj**: The type of an optional projection function. This defaults to `util::projection_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 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.

- 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 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.

**Return** 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.

```cpp
namespace hpx

namespace ranges

Functions
```

```cpp
template<typename ExPolicy, typename Rng, typename T, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_type>::difference_type>::type
count(ExPolicy&& policy, 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.

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.

**Template Parameters**
- ExPolicy: The type of the execution policy to 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 `util::projection_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: 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.

**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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_type>::difference_type>::type
count_if(
    ExPolicy&& policy,
    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.

**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.

**Return** 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 Parameters**
- **ExPolicy**: The type of the execution policy to 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 `util::projection_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:

```cpp
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.

namespace hpx

namespace ranges

Functions

template<typename ExPolicy>
util::detail::algorithm_result<ExPolicy, typename traits::range_iterator<Rng>::type>::type 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.
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.

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.

Return 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 FwdIter, typename Size>
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.
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.

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.

**Return** 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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename Pred, typename Proj1, typename Proj2>
bool ends_with(FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, Pred &&pred, Proj1 &&proj1, 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 `util::projection_identity`
- **Proj2**: The type of an optional projection function for the destination range. This defaults to `util::projection_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.

Return 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename Pred, typename Proj1, typename Proj2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with(ExPolicy &&policy, FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, Pred &&pred, Proj1 &&proj1, 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.

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 `util::projection_identity`
• Proj2: The type of an optional projection function for the destination range. This defaults to `util::projection_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 ele-
ments 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.

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.

**Return** 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.

```cpp
template<typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
bool ends_with (Rng1 &&rng1, Rng2 &&rng2, Pred &&pred, Proj1 &&proj1, 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 `util::projection_identity`
- **Proj1**: The type of an optional projection function for the destination range. This defaults to `util::projection_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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type ends_with (ExPolicy &&policy, Rng1 &&rng1, Rng2 &&rng2, Pred &&pred, Proj1 &&proj1, 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.

**Note**  Complexity: Linear: at most \( \min(N_1, N_2) \) 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 `util::projection_identity`
- **Proj2**: The type of an optional projection function for the destination range. This defaults to `util::projection_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.

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.

**Return**  The `ends_with` algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequenced_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.

```cpp
namespace hpx

namespace ranges

Functions

```cpp

template<
  typename ExPolicy,
  typename Iter1, typename Sent1,
  typename Iter2, typename Sent2,
  typename Pred = ranges::equal_to,
  typename Proj1 = util::projection_identity,
  typename Proj2 = util::projection_identity>

```
```

util::detail::algorithm_result<ExPolicy, bool>::type 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.

**Note**  Complexity: At most \( \min(last1 - first1, last2 - first2) \) 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.

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 `util::projection_identity`

Proj2: The type of an optional projection function applied to the second range. This defaults to `util::projection_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:

```cpp
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.

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  The two ranges are considered equal if, for every iterator i in the range [first1, last1), *(first2 + (i - first1)). This overload of `equal` uses `operator==` to determine if two elements are equal.

Return  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.

```cpp
template<

typename ExPolicy,

typename Rng1,

typename Rng2,

typename Pred = ranges::equal_to,

typename Proj1 = util::projection_identity,

typename Proj2 = util::projection_identity

454 Chapter 2. What's so special about HPX?
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.

**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 `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 `util::projection_identity`.
- `Proj2`: The type of an optional projection function applied to the second range. This defaults to `util::projection_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.

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** The two ranges are considered equal if, for every iterator `i` in the range `[first1,last1)`, `*(first1 + (i - first1))` equals `*(first2 + (i - first1))`. This overload of `equal` uses operator== to determine if two elements are equal.

**Return** 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.

```cpp
namespace hpx
```
namespace ranges

Functions

template<
    typename InIter,
    typename Sent,
    typename OutIter,
    typename T>

exclusive_scan_result<InIter, OutIter>::exclusive_scan(InIter first, Sent last, OutIter dest, T init)

Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, \(*\text{first}, \ldots, \ast(\text{first} + (i - \text{result}) - 1))\)

The reduce operations in the parallel exclusive_scan algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

Note Complexity: \(O(\text{last} - \text{first})\) applications of the predicate \std::plus<T>\.

Template Parameters

- \(\text{InIter}\): The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- \(\text{Sent}\): The type of the source sentinel (deduced). This sentinel type must be a sentinel for \text{InIter}.
- \(\text{OutIter}\): The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- \(\text{T}\): The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- \(\text{first}\): Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \(\text{last}\): Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- \(\text{dest}\): Refers to the beginning of the destination range.
- \(\text{init}\): The initial value for the generalized sum.

The difference between exclusive_scan and inclusive_scan is that inclusive_scan includes the \(i\)th input element in the \(i\)th sum.

Return The exclusive_scan algorithm returns util::in_out_result<InIter, OutIter>. 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.

Note GENERALIZED_NONCOMMUTATIVE_SUM(+, a_1, \ldots, a_N) is defined as:

- \(a_1\) when \(N\) is 1
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a_1, \ldots, a_K)
- GENERALIZED_NONCOMMUTATIVE_SUM(+, a_M, \ldots, a_N) where \(1 < K+1 = M <= N\).

template<
    typename ExPolicy,
    typename FwdIter1,
    typename Sent,
    typename FwdIter2,
    typename T>

util::detail::algorithm_result<ExPolicy, exclusive_scan_result<FwdIter1, FwdIter2>>::type exclusive_scan(ExPolicy &&policy,
    FwdIter1 first,
    Sent last,
    FwdIter2 dest,
    T init)

Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, init, \(*\text{first}, \ldots, \ast(\text{first} + (i - \text{result}) - 1))\)
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.

**Note** Complexity: \(O(last - first)\) applications of the predicate \(std::plus<T>\).

**Template Parameters**
- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{FwdIter1}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{Sent}: The type of the source sentinel (deduced). This sentinel type must be a sentinel for \texttt{FwdIter}.
- \texttt{FwdIter2}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{T}: The type of the value to be used as initial (and intermediate) values (deduced).

**Parameters**
- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \texttt{last}: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- \texttt{dest}: Refers to the beginning of the destination range.
- \texttt{init}: The initial value for the generalized sum.

The difference between exclusive_scan and inclusive_scan is that inclusive_scan includes the \(i\)th input element in the \(i\)th sum.

**Return** The exclusive_scan algorithm returns a \texttt{hpx::future<util::in_out_result<FwdIter1, FwdIter2>>>} if the execution policy is of type sequenced_task_policy or parallel_task_policy and returns \texttt{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.

**Note** GENERALIZED_NONCOMMUTATIVE_SUM(\(+\, a_1, \ldots, a_N\)) is defined as:
- \(a_1\) when \(N = 1\)
- GENERALIZED_NONCOMMUTATIVE_SUM(\(+\, a_1, \ldots, a_K\)) – GENERALIZED_NONCOMMUTATIVE_SUM(\(+\, a_M, \ldots, a_N\)) where \(1 < K+1 = M <= N\).

**Template Parameters**
- \texttt{InIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- \texttt{Sent}: The type of the source sentinel (deduced). This sentinel type must be a sentinel for \texttt{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).
• **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:

\[
\text{Ret fun}(\text{const Type1} &a, \text{const Type1} &b);
\]

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.

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.

**Return** The **exclusive_scan** algorithm returns \( \text{util::in_out_result<InIter, OutIter>} \). 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.

**Note** **GENERALIZED_NONCOMMUTATIVE_SUM**(\( op, a_1, \ldots, a_N \)) is defined as:

- **a_1** when \( N = 1 \)
- **op**(**GENERALIZED_NONCOMMUTATIVE_SUM**(\( op, a_1, \ldots, a_K \)), **GENERALIZED_NONCOMMUTATIVE_SUM**(\( op, a_M, \ldots, a_N \))) where \( 1 < K+1 = M <= N \).

Assigns through each iterator \( i \) in \([\text{result}, \text{result + (last - first)})\) the value of **GENERALIZED_NONCOMMUTATIVE_SUM**(binary \( op \), \( init \), \( *\text{first} \), \( \ldots \), \( *(\text{first + (i - result) - 1}) \)).

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.

**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 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.

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.

Return 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.

Note GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) is defined as:
• a1 when N is 1
• op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK), GENERAL-
IZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN)) where 1 < K+1 = M <= N.

```template<typename Rng, typename O, typename T>
exclusive_scan_result<traits::range_iterator_t<Rng>, O> exclusive_scan (Rng &rng, O dest, T init)
Assigns through each iterator i in [result, result + (last - first)) the value of GENERAL-
IZED_NONCOMMUTATIVE_SUM(+, init, *(first + i - result), *(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.

Note Complexity: O(last - first) applications of the predicate std::plus<T>.

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.
• **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.
- **init**: The initial value for the generalized sum.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the ith input element in the ith sum.

**Return**
- The `exclusive_scan` algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`.
- 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.

**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`.

```cpp
template<typename ExPolicy, typename Rng, typename O, typename T>
util::detail::algorithm_result<ExPolicy, exclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type exclusive_scan
```

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.

**Note**
- Complexity: `O(last - first)` applications of the predicate `std::plus<T>`.

**Template Parameters**
- **ExPolicy**: The type of the execution policy to 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).

**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.

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.

**Return** 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.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(+, a_1, \ldots, a_N)` is defined as:
- $a_1$ when $N$ is 1
- `GENERALIZED_NONCOMMUTATIVE_SUM(+, a_1, \ldots, a_K)`
- `GENERALIZED_NONCOMMUTATIVE_SUM(+, a_M, \ldots, a_N)` where $1 < K+1 = M \leq N$.

```cpp
template<typename Rng, typename O, typename T, typename Op>
exclusive_scan_result<traits::range_iterator_t<Rng>, O> exclusive_scan(Rng &&rng, O dest, T init, Op &&op)
```

Assigns through each iterator $i$ in `[result, result + (last - first))` the value of `GENERALIZED_NONCOMMUTATIVE_SUM(+, init, *first, \ldots, *(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.

**Note** Complexity: $O(last - first)$ applications of the predicate `std::plus<T>`.

**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.
- **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:

```cpp
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.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the ith input element in the ith sum.

**Return** The `exclusive_scan` algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(+, a_1, \ldots, a_N)` is defined as:
- $a_1$ when $N$ is 1
- `GENERALIZED_NONCOMMUTATIVE_SUM(+, a_1, \ldots, a_K)`
- `GENERALIZED_NONCOMMUTATIVE_SUM(+, a_M, \ldots, a_N)` where $1 < K+1 = M \leq N$. 

2.8. API reference 461
template<typename ExPolicy, typename Rng, typename O, typename T, typename Op>
util::detail::algorithm_result<ExPolicy, exclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type exclusive_scan(ExPolicy&& policy, Rng&& rng, O dest, T init, Op&& op)

Assigns through each iterator $i$ in $[\text{result, result + (last - first))$ 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.

**Note** Complexity: $O(last - first)$ applications of the predicate std::plus<T>.

**Template Parameters**
- ExPolicy: The type of the execution policy to 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:

```cpp
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.

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.

**Return** 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.

**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.

```cpp
namespace hpx

Functions

```cpp
template<typename ExPolicy, typename Rng, typename T>
util::detail::algorithm_result<ExPolicy>::type fill (ExPolicy &&policy, Rng &&rng, 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.

**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.
- 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.

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.

**Return** 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).

```cpp
template<typename ExPolicy, typename Iterator, typename Size, typename T>
util::detail::algorithm_result<ExPolicy, Iterator>::type fill_n (ExPolicy &&policy, 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.

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.
Note  Complexity: Performs exactly \textit{count} assignments, for count > 0.

Template Parameters
- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{Iterator}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- \texttt{Size}: The type of the argument specifying the number of elements to apply \texttt{f} to.
- \texttt{T}: The type of the value to be assigned (deduced).

Parameters
- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \texttt{count}: Refers to the number of elements starting at \texttt{first} the algorithm will be applied to.
- \texttt{value}: The value to be assigned.

The comparisons in the parallel \texttt{fill_n} algorithm invoked with an execution policy object of type \texttt{parallel_policy} or \texttt{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return  The \texttt{fill_n} algorithm returns a \texttt{hpx::future<void>} if the execution policy is of type \texttt{sequenced_task_policy} or \texttt{parallel_task_policy} and returns \texttt{difference_type} otherwise (where \texttt{difference_type} is defined by \texttt{void}).

\begin{verbatim}
namespace hpx

namespace ranges

Functions

template<
  typename \texttt{ExPolicy},
  typename \texttt{Iter},
  typename \texttt{Sent},
  typename \texttt{T},
  typename \texttt{Proj} = \texttt{util::projection_identity}>
util::detail::algorithm_result<ExPolicy, Iter>::type \texttt{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.
\end{verbatim}

The comparison operations in the parallel \texttt{find} algorithm invoked with an execution policy object of type \texttt{sequenced_policy} execute in sequential order in the calling thread.

Note  Complexity: At most last - first applications of the operator==().

Template Parameters
- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{Iter}: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{Sent}: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for \texttt{Iter}.
- \texttt{T}: The type of the value to find (deduced).
\textbf{Proj:} The type of an optional projection function. This defaults to \texttt{util::projection_identity}

\textbf{Parameters}

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first}: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- \texttt{last}: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- \texttt{val}: the value to compare the elements to
- \texttt{proj}: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate \texttt{is} invoked.

The comparison operations in the parallel \texttt{find} algorithm invoked with an execution policy object of type \texttt{parallel_policy} or \texttt{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \texttt{find} algorithm returns a \texttt{hpx::future<FwdIter>} if the execution policy is of type \texttt{sequenced_task_policy} or \texttt{parallel_task_policy} and returns \texttt{FwdIter} otherwise. The \texttt{find} algorithm returns the first element in the range \texttt{[first, last)} that is equal to \texttt{val}. If no such element in the range of \texttt{[first, last)} is equal to \texttt{val}, then the algorithm returns \texttt{last}.

\begin{verbatim}
template<typename ExPolicy, typename Rng, typename T, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, Iter>::type find(ExPolicy &&policy, Rng &&rng, T const &val, Proj &&proj = Proj())

Returns the first element in the range \texttt{[first, last)} that is equal to \texttt{val}.
\end{verbatim}

The comparison operations in the parallel \texttt{find} algorithm invoked with an execution policy object of type \texttt{sequenced_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: At most \texttt{last - first} applications of the operator\texttt{==}."

\textbf{Template Parameters}

- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \texttt{T}: The type of the value to find (deduced).
- \texttt{Proj}: The type of an optional projection function. This defaults to \texttt{util::projection_identity}

\textbf{Parameters}

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \texttt{val}: the value to compare the elements to
- \texttt{proj}: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate \texttt{is} invoked.

The comparison operations in the parallel \texttt{find} algorithm invoked with an execution policy object of type \texttt{parallel_policy} or \texttt{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \texttt{find} algorithm returns a \texttt{hpx::future<FwdIter>} if the execution policy is of type \texttt{sequenced_task_policy} or \texttt{parallel_task_policy} and returns \texttt{FwdIter} otherwise. The \texttt{find} algorithm returns the first element in the range \texttt{[first, last)} that is equal to \texttt{val}. If no such element in the range of \texttt{[first, last)} is equal to \texttt{val}, then the algorithm returns \texttt{last{
Returns the last subsequence of elements \([\text{first}_2, \text{last}_2)\) found in the range \([\text{first}_1, \text{last}_1)\) using the given predicate \(f\) to compare elements.

The comparison operations in the parallel \texttt{find\_end} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: at most \(S*(N-S+1)\) comparisons where \(S = \text{distance}(\text{first}_2, \text{last}_2)\) and \(N = \text{distance}(\text{first}_1, \text{last}_1)\).

\textbf{Template Parameters}
- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{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.
- \texttt{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 \texttt{Iter1}.
- \texttt{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.
- \texttt{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 \texttt{Iter2}.
- \texttt{Pred}: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of \texttt{replace} requires \texttt{Pred} to meet the requirements of \texttt{CopyConstructible}. This defaults to std::equal\_to\<\>.
- \texttt{Proj1}: The type of an optional projection function applied to the first sequence. This defaults to \texttt{util::projection\_identity}.
- \texttt{Proj2}: The type of an optional projection function applied to the second sequence. This defaults to \texttt{util::projection\_identity}.

\textbf{Parameters}
- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first1}: Refers to the beginning of the first sequence of elements the algorithm will be applied to.
HPX Documentation, master

- **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:

```cpp
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.

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`.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = ranges::equal_to, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_end(ExPolicy &&policy, Rng1 &&rng, 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.

**Note** Complexity: at most $S*(N-S+1)$ comparisons where $S = \text{distance}(\text{begin(rng2)}, \text{end(rng2)})$ and $N = \text{distance}(\text{begin(rng)}, \text{end(rng)})$.

**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 `util::projection_identity`
- **Proj2**: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_identity`

**Parameters**
- **policy**: The execution policy to use for the scheduling of the iterations.
- **rng**: 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:

```cpp
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.

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`.

**Return** 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 = ranges::equal_to, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity>
**util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type find_first_of**

(ExPolicy, 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.

**Note** Complexity: at most $(S*N)$ comparisons where $S = \text{distance}(\text{first2}, \text{last2})$ and $N = \text{distance}(\text{first1}, \text{last1})$.

**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 util::projection_identity and is applied to the elements in rng1.
- **Proj2**: The type of an optional projection function. This defaults to util::projection_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:

```cpp
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 `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 `iterator_t<Rng2>` before the function `op` is invoked.

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`.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = ranges::equal_to, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng1>::type>::type
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.

**Note** Complexity: at most (S*N) comparisons where \( S = \text{distance}(\text{begin}(\text{rng2}), \text{end}(\text{rng2})) \) and \( N = \text{distance}(\text{begin}(\text{rng1}), \text{end}(\text{rng1})) \).

**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 `util::projection_identity` and is applied to the elements in `rng1`.
- `Proj2`: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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.

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`.

**Return** 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.

```cpp
namespace hpx
{
    namespace ranges
    {
        // API reference
    }
}
```
Functions

template<typename InIter, typename Sent, typename F, typename Proj = util::projection_identity>
hpx::ranges::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.

Note Complexity: Applies f exactly last - first times.
If the type of first satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator.

Applies f to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

Return {last, std::move(f)} where last is the iterator corresponding to the input sentinel last.

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 util::projection_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:

\[
\text{<ignored> pred(const Type &a);} \\
\]

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.

If f returns a result, the result is ignored.

Note Complexity: Applies f exactly last - first times.
If the type of first satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator.

Return {first + count, std::move(f)}

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 util::projection_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:

```cpp
<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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Sent, typename F, typename Proj = util::projection_identity>
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.

**Note** Complexity: Applies f exactly last - first times.

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.

**Return** 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 Parameters**

- ExPolicy: The type of the execution 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 CopyConstructible.
- Proj: The type of an optional projection function. This defaults to util::projection_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:

```cpp
<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.

```cpp
template<typename Rng, typename F, typename Proj = util::projection_identity>
```
Applies \( f \) to the result of dereferencing every iterator in the given range \( \text{rng} \).

If \( f \) returns a result, the result is ignored.

**Note** Complexity: Applies \( f \) exactly \( \text{size}(\text{rng}) \) times.

If the type of \textit{first} satisfies the requirements of a mutable iterator, \( f \) may apply non-constant functions through the dereferenced iterator.

**Return** \{\texttt{std::end(rng)}, \texttt{std::move(f)}\}

**Template Parameters**
- \( \text{Rng} \): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \( \text{F} \): The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \texttt{for\_each} requires \( \text{F} \) to meet the requirements of \texttt{CopyConstructible}.
- \( \text{Proj} \): The type of an optional projection function. This defaults to \texttt{util::projection\_identity}.

**Parameters**
- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \texttt{f}: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [\textit{first}, \textit{last}). The signature of this predicate should be equivalent to:

\[
\langle \text{ignored} \rangle \texttt{pred(const Type &a)};
\]

The signature does not need to have const&. The type \textit{Type} must be such that an object of type \texttt{InIter} can be dereferenced and then implicitly converted to \textit{Type}.
- \texttt{proj}: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate \texttt{is} invoked.
If \( f \) returns a result, the result is ignored.

**Note** Complexity: Applies \( f \) exactly \( \text{size(rng)} \) times.

If the type of \( \text{first} \) 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 \( \text{for_each} \) does not return a copy of its \( \text{Function} \) 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 \( \text{sequenced_policy} \) execute in sequential order in the calling thread.

**Template Parameters**

- \( \text{ExPolicy} \): The type of the execution 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.
- \( \text{Rng} \): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \( \text{F} \): The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \( \text{for_each} \) requires \( F \) to meet the requirements of \( \text{CopyConstructible} \).
- \( \text{Proj} \): The type of an optional projection function. This defaults to \( \text{util::projection_identity} \)

**Parameters**

- \( \text{policy} \): The execution policy to use for the scheduling of the iterations.
- \( \text{rng} \): Refers to the sequence of elements the algorithm will be applied to.
- \( \text{f} \): Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \( [\text{first}, \text{last}) \). The signature of this predicate should be equivalent to:

\[
\text{pred(const Type \&a)};
\]

The signature does not need to have \( \text{const} \& \). The type \( \text{Type} \) must be such that an object of type \( \text{InIter} \) can be dereferenced and then implicitly converted to \( \text{Type} \).
- \( \text{proj} \): 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 application of function objects in parallel algorithm invoked with an execution policy object of type \( \text{parallel_policy} \) or \( \text{parallel_task_policy} \) are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** The \( \text{for_each} \) algorithm returns a \( \text{hpx::future<FwdIter>} \) if the execution policy is of type \( \text{sequenced_task_policy} \) or \( \text{parallel_task_policy} \) and returns \( \text{FwdIter} \) otherwise. It returns \( \text{last} \).

```cpp
.template<typename ExPolicy, typename FwdIter, typename Size, typename F, typename Proj = util::projection_identity
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 \( [\text{first}, \text{first} + \text{count}) \), starting from \( \text{first} \) and proceeding to \( \text{first} + \text{count} - 1 \).

If \( f \) returns a result, the result is ignored.

**Note** Complexity: Applies \( f \) exactly \( \text{count} \) times.

If the type of \( \text{first} \) 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 \( \text{for_each} \) does not return a copy of its \( \text{Function} \) parameter, since parallelization may not permit efficient state accumulation.
Return 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 Parameters
- `ExPolicy`: The type of the execution 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 `CopyConstructible`.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_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.

namespace hpx

namespace ranges

Functions

```cpp
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`.

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 \([\text{first}, \text{last})\) should expose a signature equivalent to:

\[
\text{<ignored>} \ \text{pred}(\text{Iter const} & \ a, \ ...);
\]

The signature does not need to have \text{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.

**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 \(\text{last} - \text{first}\).

The first element in the input sequence is specified by \text{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:

**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.

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.

\[
\text{template<}\text{typename } \text{ExPolicy}, \text{typename } \text{Iter}, \text{typename } \text{Sent}, \text{typename } ... \text{Args} >
\]

\[\text{util::detail::algorithm_result<ExPolicy>::type } \text{for_loop} (\text{ExPolicy } && \text{policy}, \ \text{Iter } \text{first}, \ \text{Sent } \text{last}, \ \text{Args} && ... \text{args})\]

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**.

**Template Parameters**

- **ExPolicy**: The type of the execution 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 (forward 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.
- **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:

\[
\text{<ignored> pred(Iter const\& a, ...);}\]

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 algorithm, representing their current values.

**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 \( \text{first} \). 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:

**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.

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.

**Return** The for_loop algorithm returns a \( 	ext{hpx::future<void>} \) if the execution policy is of type \( 	ext{hpx::execution::sequenced_task_policy} \) or \( 	ext{hpx::execution::parallel_task_policy} \) and returns void otherwise.

```cpp
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 \( 	ext{hpx::execution::seq} \) as the execution policy.

**Requires**: \( 	ext{Rng::iterator} \) 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.

**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 the 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:

```cpp
<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.

**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:

**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.

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.

```cpp
template<
    typename ExPolicy,
    typename Rng,
    typename ... Args>
util::detail::algorithm_result<ExPolicy>::type for_loop(
    ExPolicy &&policy,
    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.

**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`.

**Template Parameters**

• **ExPolicy**: The type of the execution 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.
• **Args**: A parameter pack, its 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 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 this 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:

```cpp
<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.

**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:

**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.

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.

**Return** 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.

```cpp
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`. 

480 Chapter 2. What’s so special about HPX?
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, its 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:

```cpp
<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.

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:

**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.

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.

```cpp
template<typename ExPolicy, typename Iter, typename Sent, typename S, typename... 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.

**Template Parameters**
- **ExPolicy**: The type of the execution 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:

```cpp
<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.

**Effects**: Applies \( f \) to each element in the input sequence, with additional arguments corresponding to the reductions and inductions in the \( \text{args} \) parameter pack. The length of the input sequence is \( \text{last} - \text{first} \).

The first element in the input sequence is specified by \( \text{first} \). Each subsequent element is generated by incrementing the previous element.

Along with an element from the input sequence, for each member of the \( \text{args} \) parameter pack excluding \( f \), an additional argument is passed to each application of \( f \) as follows:

- **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.

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.

**Return** 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.

```cpp
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`.

**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 the 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 `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:

```cpp
<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.

**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:

- **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.

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.

```cpp
template<typename ExPolicy, typename Rng, typename S, typename ..., Args>
util::detail::algorithm_result<ExPolicy>::type 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 al-
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.

**Template Parameters**
- ExPolicy: The type of the execution 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 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:

```c++
<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.

**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:

**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.

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.
Return 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.

namespace hpx

namespace ranges

Functions

template<typename ExPolicy, typename Rng, typename F>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type
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.

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.

The assignments in the parallel generate algorithm invoked with an execution policy object of type sequenced_policy or parallel_task_policy are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return 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>
util::detail::algorithm_result<ExPolicy, Iter>::type
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.

**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:

```cpp
Ret fun();
```

The type Ret must be such that an object of type FwdIter can be dereferenced and assigned a value of type Ret.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Size, typename F>
util::detail::algorithm_result<ExPolicy, FwdIter>::type 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.

**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.
- 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

\[
\text{Ret} \ f() ;
\]

The type \( \text{Ret} \) must be such that an object of type \( \text{OutputIt} \) can be dereferenced and assigned a value of type \( \text{Ret} \).

The assignments in the parallel \( \text{generate}_n \) algorithm invoked with an execution policy object of type \( \text{parallel\_policy} \) or \( \text{parallel\_task\_policy} \) are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** The \( \text{replace\_if} \) algorithm returns a \( \text{hpx}::\text{future}<\text{FwdIter}> \) if the execution policy is of type \( \text{sequenced\_task\_policy} \) or \( \text{parallel\_task\_policy} \) and returns \( \text{FwdIter} \) otherwise. It returns \( \text{last} \).

```cpp
namespace hpx

namespace ranges

Functions

template<typename \text{ExPolicy}, typename \text{Iter1}, typename \text{Sent1}, typename \text{Iter2}, typename \text{Sent2}, typename \text{Pred} = \text{detail}::\text{less}, typename \text{Proj1} = \text{util}::\text{projection}\_\text{identity}, typename \text{Proj2} = \text{util}::\text{projection}\_\text{identity}>

\text{util}::\text{detail}::\text{algorithm}\_\text{result}<\text{ExPolicy}, \text{bool}>::\text{type} \text{includes} (\text{ExPolicy} && \text{policy}, \text{Iter1} \text{first1}, \text{Sent1} \text{last1}, \text{Iter2} \text{first2}, \text{Sent2} \text{last2}, \text{Pred} && \text{op} = \text{Pred}(), \text{Proj1} && \text{proj1} = \text{Proj1}(), \text{Proj2} && \text{proj2} = \text{Proj2}())
```

Returns true if every element from the sorted range \([\text{first2}, \text{last2})\) is found within the sorted range \([\text{first1}, \text{last1})\). Also returns true if \([\text{first2}, \text{last2})\) is empty. The version expects both ranges to be sorted with the user supplied binary predicate \( f \).

The comparison operations in the parallel \( \text{includes} \) algorithm invoked with an execution policy object of type \( \text{sequenced\_policy} \) execute in sequential order in the calling thread.

**Note** At most \( 2^*(N1+N2-1) \) comparisons, where \( N1 = \text{std}::\text{distance}({\text{first1}}, \text{last1}) \) and \( N2 = \text{std}::\text{distance}({\text{first2}}, \text{last2}) \).

**Template Parameters**

- \( \text{ExPolicy} \): The type of the execution policy to 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.
- \( \text{Iter1} \): The type of the source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an forward iterator.
- \( \text{Sent1} \): The type of the end source iterators used (deduced) representing the first sequence. This iterator type must meet the requirements of an sentinel for \( \text{Iter1} \).
- \( \text{Iter2} \): The type of the source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an forward iterator.
- \( \text{Sent2} \): The type of the end source iterators used (deduced) representing the second sequence. This iterator type must meet the requirements of an sentinel for \( \text{Iter2} \).
- \( \text{Pred} \): The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of \( \text{includes} \) requires \( \text{Pred} \) to meet the requirements of \( \text{CopyConstructible} \). This defaults to \( \text{std}::\text{less}<> \)
- \( \text{Proj1} \): The type of an optional projection function applied to the first sequence. This defaults to \( \text{util}::\text{projection}\_\text{identity} \)
• Proj2: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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:

```cpp
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.

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.

### Return

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. The version expects both ranges to be sorted with the user supplied binary predicate `f`.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred = detail::less, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type 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.

### Note

At most $2*(N1+N2-1)$ comparisons, where $N1 = \text{std::distance(first1, last1)}$ and $N2 = \text{std::distance(first2, last2)}$.

### 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 util::projection_identity
- **Proj2**: The type of an optional projection function applied to the second sequence. This defaults to util::projection_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:

```cpp
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.

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.

**Return** 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.

### namespace hpx

### namespace ranges

#### Functions

```cpp
template<typename InIter, typename Sent, typename OutIter>
inclusive_scan_result<InIter, OutIter> inclusive_scan(InIter first, Sent 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.

**Note** Complexity: \(O(last - first)\) applications of the predicate op.

### 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.

### Parameters

- **first**: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- **last**: Refers to sentinal value denoting the end of the sequence of elements the algorithm will be applied.
- **dest**: Refers to the beginning of the destination range.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the ith input element in the ith sum.

**Return** 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.

**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.
- **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.

### 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.
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.

**Return** 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.

**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).

**Parameters**
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

**Note** Complexity: `O(last - first)` applications of the predicate `op`.

**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).

**Parameters**
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the ith input element in the ith sum.

**Return** The `inclusive_scan` algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`.

**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**
```cpp
template<typename Rng, typename O>
inclusive_scan_result<traits::range_iterator_t<Rng>, O> inclusive_scan(Rng &&rng, O dest)
```
Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(+, \(*\text{first}, \ldots, \*(\text{first} + (i - \text{result})))\).

The reduce operations in the parallel \textit{inclusive_scan} algorithm invoked with an execution policy object of type \textit{sequenced_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: \(O(\text{last} - \text{first})\) applications of the predicate \( \text{op} \).

\textbf{Template Parameters}

- \textbf{ExPolicy}: The type of the execution policy to 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.
- \textbf{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- \textbf{O}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

\textbf{Parameters}

- \textbf{policy}: The execution policy to use for the scheduling of the iterations.
- \textbf{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \textbf{dest}: Refers to the beginning of the destination range.

The reduce operations in the parallel \textit{inclusive_scan} algorithm invoked with an execution policy object of type \textit{parallel_policy} or \textit{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

The difference between \textit{exclusive_scan} and \textit{inclusive_scan} is that \textit{inclusive_scan} includes the \( i \)th input element in the \( i \)th sum.

\textbf{Return} The \textit{inclusive_scan} algorithm returns a \textit{hpx::future<util::in_out_result<traits::range_iterator_t<Rng>, O>}} if the execution policy is of type \textit{sequenced_task_policy} or \textit{parallel_task_policy} and returns \textit{util::in_out_result<traits::range_iterator_t<Rng>, O>} otherwise. The \textit{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.

\textbf{Note} GENERALIZED_NONCOMMUTATIVE_SUM(+, a1, \ldots, aN) is defined as:
- \( a1 \) when \( N = 1 \)
- \( \text{GENERALIZED_NONCOMMUTATIVE_SUM}(+, a1, \ldots, aK) \)  
  \( = \text{GENERALIZED_NONCOMMUTATIVE_SUM}(+, aM, \ldots, aN) \) where \( 1 < K+1 = M \leq N \).

\textbf{template<typename InIter, typename Sent, typename OutIter, typename Op>}

\textbf{inclusive_scan_result<InIter, OutIter> inclusive_scan (InIter first, Sent last, OutIter dest, Op \&\&op)}

Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, \(*\text{first}, \ldots, \*(\text{first} + (i - \text{result})))\).

The reduce operations in the parallel \textit{inclusive_scan} algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

\textbf{Note} Complexity: \(O(\text{last} - \text{first})\) applications of the predicate \( \text{op} \).

\textbf{Template Parameters}

- \textbf{InIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- \textbf{Sent}: The type of the source sentinel (deduced). This sentinel type must be a sentinel for \textit{InIter}.
- \textbf{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:

\[ \text{Ret} \ fun(\text{const} \ Type1 \ &a, \ \text{const} \ Type1 \ &b) \];

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.

The difference between exclusive_scan and inclusive_scan is that inclusive_scan includes the ith input element in the ith sum.

Return 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.

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.

Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) 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.

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 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
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:

```cpp
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.

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 reduce operations in the parallel `inclusive_scan` algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

**Return** 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 `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.

**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`.

```cpp
template<typename Rng, typename O, typename Op>
inclusive_scan_result<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.

**Note** Complexity: `O(last - first)` applications of the predicate `op`.

**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:

```cpp
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.

The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the ith input element in the ith sum.

**Return** The `inclusive_scan` algorithm returns `util::in_out_result<traits::range_iterator_t<Rng>, O>`.

**Note** `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.

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.

```cpp
template<typename ExPolicy, typename Rng, typename O, typename Op>
util::detail::algorithm_result<ExPolicy, inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type inclusive_scan(ExPolicy&& policy, 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 with an execution policy object of type `sequenced_policy` execute in sequential order in the calling 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 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:

```cpp
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.

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. 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.

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<typename InIter, typename Sent, typename OutIter, typename T, typename Op>
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.

Note Complexity: O(last - first) applications of the predicate op.

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.
The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the \( i \)th input element in the \( i \)th sum. If \( op \) is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

**Return** 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.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aN)` is defined as:
- \( a1 \) when \( N \) is 1
- \( \text{op}(\text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, a1, \ldots, aK), \text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, aM, \ldots, aN)) \) where \( 1 < K+1 = M \leq N \).

```
template<
    typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2,
    typename T, typename Op
>
util::detail::algorithm_result<
    ExPolicy
>::type
inclusive_scan(
    ExPolicy&& policy,
    FwdIter1 first,
    Sent last,
    FwdIter2 dest,
    T init,
    Op&& op)
```

Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, \ldots, *(first + (i - \text{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.

**Note** Complexity: \( O(\text{last} - \text{first}) \) applications of the predicate \( op \).

**Template Parameters**
- \( \text{ExPolicy} \): The type of the execution policy to 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.
- \( \text{FwdIter1} \): The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \( \text{Sent} \): The type of the source sentinel (deduced). This sentinel type must be a sentinel for \( \text{InIter} \).
- \( \text{FwdIter2} \): The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- \( \text{T} \): The type of the value to be used as initial (and intermediate) values (deduced).
- \( \text{Op} \): The type of the binary function object used for the reduction operation.

**Parameters**
- \( \text{policy} \): The execution policy to use for the scheduling of the iterations.
- \( \text{first} \): Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \( \text{last} \): Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- \( \text{dest} \): Refers to the beginning of the destination range.
- \( \text{init} \): The initial value for the generalized sum.
- \( \text{op} \): 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:

```cpp
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.

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 *i*th input element in the *i*th sum. If `op` is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

**Return** 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.

**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<typename Rng, typename O, typename Op, typename T>
inclusive_scan_result<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.

**Note** Complexity: `O(last - first)` applications of the predicate `op`.

**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:

```cpp
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.
The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i\(^{th}\) input element in the i\(^{th}\) sum. If \(op\) is not mathematically associative, the behavior of `inclusive_scan` may be non-deterministic.

**Return** 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.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \ldots, a_N)` is defined as:
- \(a_1\) when \(N\) is 1
- \(op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a_1, \ldots, a_K), GENERALIZED_NONCOMMUTATIVE_SUM(op, a_M, \ldots, a_N))\) where \(1 < K+1 = M \leq N\).

```cpp
template<typename ExPolicy, typename Rng, typename O, typename Op, typename T>
util::detail::algorithm_result<ExPolicy, inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type
inclusive_scan(ExPolicy&& policy, Rng&& rng, O dest, Op&& op, T init)
```

Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, \text{init}, *\text{first}, \ldots, *(\text{first} + (i - \text{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.

**Note** Complexity: \(O(\text{last} - \text{first})\) 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 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:

```cpp
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. 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.

Return 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.

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.

namespace hpx

Functions

template<typename ExPolicy, typename Iter, typename Sent, typename Comp = detail::less, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, bool>::type 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.

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 util::projection_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.
• **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.

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.

**Return** 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. 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**

- **ExPolicy**: The type of the execution policy to 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 `util::projection_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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Iter, typename Sent, typename Comp = detail::less, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, Iter>::type is_heap_until(
    ExPolicy &&policy,
    Iter first,
    Sent sent,
    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**
- `ExPolicy`: The type of the execution policy to 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 `util::projection_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.
- `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.

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.
Return 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 Rng, typename Comp = detail::less, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type 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.

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 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 `util::projection_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.

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.

2.8. API reference 503
Return 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.

namespace hpx

namespace ranges

Functions

template<typename FwdIter, typename Sent, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_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 re-
requirements of a forward iterator.
- `Pred`: The type of an optional function/function object to use.
- `Proj`: The type of an optional projection function. This defaults to `util::projection_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

```cpp
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.

Return 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::v1::detail::less, typename Proj = hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type>

```cpp
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type 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.

**Note** Complexity: at most (N+S-1) comparisons where \( N = \text{distance}(\text{first}, \text{last}) \). \( S = \text{number of partitions} \)

**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\
- **Proj**: The type of an optional projection function. This defaults to *util::projection_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

```cpp
type::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.

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.

**Return** 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.

```cpp
template<typename Rng, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>
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 executes in sequential order in the calling thread.

**Note** Complexity: at most (N+S-1) comparisons where \( N = \text{size}(\text{rng}) \). \( S = \text{number of partitions} \)

**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 *util::projection_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

```cpp
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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename Pred = hpx::parallel::v1::detail::less, typename Proj = hpx::parallel::util::projection_identity>

hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type 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.

**Note** Complexity: at most \((N+S-1)\) comparisons where \(N = \text{size}(rng)\). \(S = \text{number of partitions}\)

**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 `CopyConstructible`. This defaults to `std::less<>
- **Proj**: The type of an optional projection function. This defaults to `util::projection_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

```cpp
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.

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.
Return 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.

```
return hpx::future<bool>(is_sorted(rng, predicate));
```

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.
- Proj: The type of an optional projection function. This defaults to util::projection_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

```c++
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.

Return 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.

```
FwdIter is_sorted_until(FwdIter first, Sent last, Pred && pred, Proj && proj = Proj())
```

The result is that the first unsorted element is returned.
The comparison operations in the parallel \texttt{is\_sorted\_until} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} executes in sequential order in the calling thread.

\textbf{Note} Complexity: at most (N+S-1) comparisons where \( N = \text{distance(first, last)} \). \( S = \) number of partitions

\textbf{Template Parameters}

- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{FwdIter}: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- \texttt{Pred}: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of \texttt{is\_sorted\_until} requires \texttt{Pred} to meet the requirements of \texttt{CopyConstructible}. This defaults to \texttt{std::less<>
- \texttt{Proj}: The type of an optional projection function. This defaults to \texttt{util::projection\_identity}

\textbf{Parameters}

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{first}: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- \texttt{last}: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- \texttt{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

\begin{verbatim}
bool pred(const Type &a, const Type &b);
\end{verbatim}

The signature does not need to have \texttt{const} \&, but the function must not modify the objects passed to it. The type \texttt{Type} must be such that objects of types \texttt{FwdIter} can be dereferenced and then implicitly converted to \texttt{Type}.

- \texttt{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 comparison operations in the parallel \texttt{is\_sorted\_until} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \texttt{is\_sorted\_until} algorithm returns a \texttt{hpx::future\<FwdIter\>} if the execution policy is of type \texttt{task\_execution\_policy} and returns \texttt{FwdIter} otherwise. The \texttt{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.

\[
\text{template<typename } \text{Rng, typename Pred } = \text{ hpx::parallel::v1::detail::less, typename Proj } = \text{ hpx::parallel::util::projection_identity, typename ExPolicy } = \text{ hpx::traits::range_iterator}<\text{Rng}>::\text{type}\]
\[
\text{is_sorted_until} \text{ (ExPolicy } \&\&\text{policy, Rng } \&\&\text{rng, }
\quad \text{Pred } \&\&\text{pred } = \text{Pred()}, \text{Proj } \&\&\text{proj } = \text{Proj()})
\]

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 \textit{is_sorted_until} algorithm execute in sequential order in the calling thread.  

\textbf{Note} Complexity: at most (N+S-1) comparisons where } N = \text{size(rng). } S = \text{number of partitions}

\textbf{Template Parameters}

- \text{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- \text{Pred}: The type of an optional function/function object to use.
- \text{Proj}: The type of an optional projection function. This defaults to \text{util::projection_identity}

\textbf{Parameters}

- \text{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \text{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

\[
\text{bool pred(const Type } \&a, \text{ const Type } \&b);\]

The signature does not need to have \text{const } \&, but the function must not modify the objects passed to it. The type \text{Type} must be such that objects of types \text{FwdIter} can be dereferenced and then implicitly converted to \text{Type}.

- \text{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.

\textbf{Return} The \textit{is_sorted_until} algorithm returns a \text{FwdIter}. The \textit{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.

\[
\text{template<typename } \text{ExPolicy, typename } \text{Rng, typename Pred } = \text{ hpx::parallel::v1::detail::less, typename Proj } = \text{ hpx::parallel::util::detail::algorithm_result<ExPolicy, typenam}
\quad \text{e hpx::traits::range_iterator<Rng>::type}\cdot\text{type}\] is_sorted_until
\[
\text{ExPolicy } \&\&\text{policy, Rng } \&\&\text{rng, }
\quad \text{Pred } \&\&\text{pred } = \text{Pred()}, \text{Proj } \&\&\text{proj } = \text{Proj()})
\]

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 \textit{is_sorted_until} algorithm invoked with an execution policy
object of type *sequenced_policy* executes in sequential order in the calling thread.

**Note** Complexity: at most (N+S-1) comparisons where \( N = \text{size}(\text{rng}) \). \( S = \text{number of partitions} \)

**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 `util::projection_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

```cpp
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.

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.

**Return** 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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename InIter1, typename Sent1, typename InIter2, typename Sent2, typenameProj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity, typename Pred = 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 with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

**Note** Complexity: At most \( 2 \times \min(N1, N2) \) applications of the comparison operation, where \( N1 = \text{std::distance}(\text{first1}, \text{last}) \) and \( N2 = \text{std::distance}(\text{first2}, \text{last2}) \).

**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 `util::projection_identity`.
• **Proj2**: The type of an optional projection function for `FwdIter2`. This defaults to `util::projection_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.

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** 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

**Return** 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.`
parallel::util::detail::algorithm_result<ExPolicy, bool>::type 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.

**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).

**Template Parameters**

- **ExPolicy**: The type of the execution policy to 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 util::projection_identity
- **Proj2**: The type of an optional projection function for FwdIter2. This defaults to util::projection_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.

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** 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

**Return** The `lexicographical_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 `lexicographical_compare` algorithm returns true if the first range is lexicographically less, otherwise it returns false. range `[first2, last2)`, it returns false.

```cpp
template<typename Rng1, typename Rng2, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity, typename Pred = std::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 $2 \times \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std::distance}(\text{std::begin(rng1), std::end(rng1)})$ and $N2 = \text{std::distance}(\text{std::begin(rng2), std::end(rng2)})$.

**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 `util::projection_identity`
• `Proj2`: The type of an optional projection function for elements of the second range. This defaults to `util::projection_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.

**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**.

**Return** The `lexicographical_compare` algorithm returns `bool`. The `lexicographical_compare` algorithm returns true if the first range is lexicographically less, otherwise it returns false. range (first2, last2), it returns false.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity, typename Pred = detail::less>
parallel::util::detail::algorithm_result<ExPolicy, bool>::type 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.

**Note** Complexity: At most $2 \times \min(N1, N2)$ applications of the comparison operation, where $N1 = \text{std::distance}($std::begin(rng1), std::end(rng1))$ and $N2 = \text{std::distance}($std::begin(rng2), std::end(rng2)).

**Template Parameters**
- **ExPolicy**: The type of the execution policy to 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 `util::projection_identity`
• **Proj2**: The type of an optional projection function for elements of the second range. This defaults to `util::projection_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.

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** 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

**Return** 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.

```cpp
namespace hpx
{

namespace ranges
{

}```
Functions

template<typename ExPolicy, typename Rng, typename Comp, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type
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.

**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 util::projection_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

```cpp
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.

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.

**Return** 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 = util::projection_identity>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type make_heap(
    ExPolicy &&policy,
    Rng &&rng,
    Proj &&proj
    = Proj{});

Constructs a max heap in the range [first, last). Uses the operator < for comparisons.

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.

**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 util::projection_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.

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.

**Return** The make_heap algorithm returns a hpx::future<void> if the execution policy is of type task_execution_policy and returns void otherwise.

namespace hpx

namespace ranges

**Functions**

template<typename ExPolicy, typename Iter1, typename Sent, typename Iter2, typename Sent2, typename Iter3, typename Comp = hpx::ranges::less, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity>
Merges two sorted ranges \([first_1, last_1)\) and \([first_2, last_2)\) into one sorted range beginning at \(dest\). The order of equivalent elements in each of the 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 \textit{merge} algorithm invoked with an execution policy object of type \textit{sequenced\_policy} execute in sequential order in the calling thread.

**Note** Complexity: Performs \(O(\text{std::distance}(first_1, last_1) + \text{std::distance}(first_2, last_2))\) 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 a 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 a 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 a random access iterator.
- **Comp**: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \textit{merge} requires \(Comp\) to meet the requirements of \textit{Copy\_Constructible}. This defaults to \textit{std::less<\rangle}.
- **Proj1**: The type of an optional projection function to be used for elements of the first range. This defaults to \textit{util::projection\_identity}.
• **Proj2**: The type of an optional projection function to be used for elements of the second range. This defaults to `util::projection_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:

```cpp
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`.
- **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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename RandIter3, typename Comp = hpx::range_v3::less>
```
Merges two sorted ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) into one sorted range beginning at \(\text{dest}\). 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 \textit{merge} algorithm invoked with an execution policy object of type \textit{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Performs \(O(\text{std}::\text{distance}(\text{first1}, \text{last1}) + \text{std}::\text{distance}(\text{first2}, \text{last2}))\) applications of the comparison \textit{comp} and the each projection.

\textbf{Template Parameters}
- \textit{ExPolicy}: The type of the execution policy to 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.
- \textit{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.
- \textit{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.
- \textit{RandIter3}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an random access iterator.
- \textit{Comp}: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \textit{merge} requires \textit{Comp} to meet the requirements of \textit{CopyConstructible}. This defaults to \textit{std::less<>
- \textit{Proj1}: The type of an optional projection function to be used for elements of the first range. This defaults to \textit{util::projection\_identity}
- \textit{Proj2}: The type of an optional projection function to be used for elements of the second range. This defaults to \textit{util::projection\_identity}

\textbf{Parameters}
- \textit{policy}: The execution policy to use for the scheduling of the iterations.
- \textit{rng1}: Refers to the first range of elements the algorithm will be applied to.
- \textit{rng2}: Refers to the second range of elements the algorithm will be applied to.
- \textit{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:

```cpp
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.

• 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.

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.

**Return** The merge algorithm returns a `hpx::future<merge_result<RandIter1, RandIter2, RandIter3>>` if the execution policy is of type sequenced_task_policy or parallel_task_policy and returns `merge_result<RandIter1, RandIter2, RandIter3>` 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.

```cpp
template<typename ExPolicy, typename Iter, typename Sent, typename Comp = hpx::ranges::less, typename Proj = util::detail::algorithm_result<ExPolicy, Iter>::type
 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.

**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 util::projection_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:

  ```cpp
  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`.

- **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 `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.

**Return** 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`.

```cpp
template<typename ExPolicy, typename Rng, typename RandIter, typename Comp = hpx::ranges::less, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, RandIter>::type inplace_merge(ExPolicy &&policy,
    Rng &&rng, RandIter 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.

**Note** Complexity: Performs $O(\text{std::distance(first, last)})$ 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.
- **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<`
- **Proj**: The type of an optional projection function. This defaults to `util::projection_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:

```c++
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.

- `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 `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.

**Return** The inplace_merge 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 inplace_merge algorithm returns the source iterator last.

```c++
namespace hpx

namespace parallel

Functions

```c++
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type
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.

**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 util::projection_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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename Proj = util::projection_identity, typename F = detail::less>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type 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.

**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 `util::projection_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 left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

\[
\text{bool } \text{pred}(\text{const } \text{Type1 } &a, \text{ const } \text{Type1 } &b);
\]

The signature does not need to have const &, but the function must not modify the objects passed to it. The type \( \text{Type1} \) must be such that objects of type \( \text{FwdIter} \) can be dereferenced and then implicitly converted to \( \text{Type1} \).

• \( \text{proj} \): 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 comparisons in the parallel \emph{max_element} algorithm invoked with an execution policy object of type \emph{parallel_policy} or \emph{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** The \emph{max_element} algorithm returns a \( \text{hpx::future<\text{FwdIter}>} \) if the execution policy is of type \emph{sequenced_task_policy} or \emph{parallel_task_policy} and returns \( \text{FwdIter} \) otherwise. The \emph{max_element} algorithm returns the iterator to the smallest element in the range \([\text{first}, \text{last})\). 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 \text{ExPolicy}, typename \text{Rng}, typename \text{Proj} = \text{util::projection_identity}, typename \text{F} = \text{detail::less}>
\text{util::detail::algorithm_result<ExPolicy, \text{hpx::util::tagged_pair<\text{tag::min}, \text{typename hpx::traits::range_traits<\text{Rng}>::iterator_type>>, \text{max}, \text{typename hpx::traits::range_traits<\text{Rng}>::iterator_type>>>}}\text{minmax_element}\text{ExPolicy}\&\& \text{policy}, \text{Rng}\&\& \text{rng}, \text{F}\&\& \text{f} = \text{F}(), \text{Proj}\&\& \text{proj} = \text{Proj}()\text{Finds the greatest element in the range [first, last) using the given comparison function f.}
```

The comparisons in the parallel \emph{minmax_element} algorithm invoked with an execution policy object of type \emph{sequenced_policy} execute in sequential order in the calling thread. **Note** Complexity: At most \( \text{max(floor(3/2*(N-1)), 0)} \) applications of the predicate, where \( N = \text{std::distance(first, last)} \).

**Template Parameters**

• \( \text{ExPolicy} \): The type of the execution policy to 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.

• \( \text{Rng} \): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.

• \( \text{F} \): The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \emph{minmax_element} requires \( \text{F} \) to meet the requirements of \emph{CopyConstructible}.

• \( \text{Proj} \): The type of an optional projection function. This defaults to \emph{util::projection_identity}

**Parameters**

• \( \text{policy} \): The execution policy to use for the scheduling of the iterations.

• \( \text{rng} \): Refers to the sequence of elements the algorithm will be applied to.

• \( f \): The binary predicate which returns true if the left argument is less than the right element. This argument is optional and defaults to \emph{std::less}. The signature of the predicate function should be equivalent to the following:

\[
\text{bool } \text{pred}(\text{const } \text{Type1 } &a, \text{ const } \text{Type1 } &b);
\]

The signature does not need to have const &, but the function must not modify the objects passed to it. The type \( \text{Type1} \) must be such that objects of type \( \text{FwdIter} \) can be dereferenced and then implicitly converted to \( \text{Type1} \).
• **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.

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.

**Return** The `minmax_element` algorithm returns a `hpx::future<tagged_pair<tag::min(FwdIter), tag::max(FwdIter)>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `tagged_pair<tag::min(FwdIter), tag::max(FwdIter)>>` otherwise. The `minmax_element` algorithm returns a pair consisting of an iterator to the smallest element as the first element and an iterator to the greatest element as the second. 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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Pred = ranges::equal_to, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity>
util::detail::algorithm_result<ExPolicy, ranges::mismatch_result<FwdIter1, FwdIter2>>::type
mismatch(
    ExPolicy&& policy,
    FwdIter1 first1, FwdIter1 last1,
    FwdIter2 first2, FwdIter2 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.

**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.
Template Parameters

- **ExPolicy**: The type of the execution policy to 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 `util::projection_identity`.
- **Proj2**: The type of an optional projection function applied to the second range. This defaults to `util::projection_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:

  ```cpp
  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.

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** 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.

**Return** 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.
util::detail::algorithm_result<ExPolicy, ranges::mimatch_result<FwdIter1, FwdIter2>>::type mismatch (ExPolicy &&policy, Rng1 &&rng1, Rng2 &&rng2, Pred &&op = Pred(), Proj1 &&proj1 = Proj1(), Proj2 &&proj2 = Proj2())

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.

**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 util::projection_identity.
- **Proj2**: The type of an optional projection function applied to the second range. This defaults to util::projection_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:

```cpp
def 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.
• \texttt{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 \textit{is} invoked.

The comparison operations in the parallel \texttt{mismatch} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \texttt{mismatch} algorithm returns a \texttt{hpx::future<std::pair<FwdIter1, FwdIter2> >} if the execution policy is of type \texttt{sequenced\_task\_policy} or \texttt{parallel\_task\_policy} and returns \texttt{std::pair<FwdIter1, FwdIter2>} otherwise. The \texttt{mismatch} algorithm returns the first mismatching pair of elements from two ranges: one defined by \texttt{[first1, last1)} and another defined by \texttt{[first2, last2)}.

\begin{verbatim}
namespace hpx

Functions

template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter>
util::detail::algorithm_result<ExPolicy, ranges::move_result<FwdIter1, FwdIter>>::type move(ExPolicy &&policy, FwdIter1 iter, Sent1 sent, FwdIter dest)

Moves the elements in the range \texttt{rng} to another range beginning at \texttt{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 \texttt{copy} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Performs exactly std::distance(begin(rng), end(rng)) assignments.

\textbf{Template Parameters}

• \texttt{ExPolicy}: The type of the execution policy to 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.

• \texttt{FwdIter1}: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

• \texttt{Sent1}: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for \texttt{FwdIter1}.

• \texttt{FwdIter}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

\textbf{Parameters}

• \texttt{policy}: The execution policy to use for the scheduling of the iterations.

• \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.

• \texttt{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 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.

Return 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 FwdIter>
util::detail::algorithm_result<ExPolicy, ranges::move_result<
  typename hpx::traits::range_traits<Rng>::iterator_type, FwdIter>>::type
move(ExPolicy&& policy, Rng&& rng, FwdIter 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.

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.

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.

Return 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

530 Chapter 2. What’s so special about HPX?
of the input iterator \(last\) and the output iterator to the element in the destination range, one past the last element moved.

```cpp
namespace hpx

namespace ranges

Functions

template<typename RandomIt, typename Sent, typename Pred, typename Proj>
RandomIt nth_element(RandomIt first, RandomIt nth, Sent last, Pred &&pred, Proj &&proj)
```

\(\text{n}\text{th}\_\text{element}\) is a partial sorting algorithm that rearranges elements in \([\text{first}, \text{last})\) such that the element pointed at by \(\text{nth}\) is changed to whatever element would occur in that position if \([\text{first}, \text{last})\) were sorted and all of the elements before this new \(\text{nth}\) element are less than or equal to the elements after the new \(\text{nth}\) element.

The comparison operations in the parallel \(\text{n}\text{th}\_\text{element}\) 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 \(N = \text{last} - \text{first}\).

**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 util::projection_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.
- **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:

```cpp
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 projection_identity.

**Return** The \(\text{n}\text{th}\_\text{element}\) algorithm returns returns RandomIt. The \(\text{n}\text{th}\_\text{element}\) algorithm returns an iterator equal to last.

```cpp
template<typename ExPolicy, typename RandomIt, typename Sent, typename Pred, typename 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.

**Note** Complexity: Linear in \( \text{std::distance}(\text{first}, \text{last}) \) on average. \( O(N) \) applications of the predicate, and \( O(N \log N) \) swaps, where \( N = \text{last} - \text{first} \).

**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 `util::projection_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:

  ```cpp
  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 `projection_identity`.

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.

**Return** 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.

```cpp
template<typename Rng, typename Pred, typename 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 util::projection_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:

```cpp
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 projection_identity.

**Return** The nth_element algorithm returns returns hpx::traits::range_iterator_t<Rng>. The nth_element algorithm returns an iterator equal to last.

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.

**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 `util::projection_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:

```cpp
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 `projection_identity`.

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.

**Return** 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.

```cpp
namespace hpx

namespace ranges

Functions

```cpp
template<typename RandomIt, typename Sent, typename Comp, typename Proj>
RandomIt partial_sort (RandomIt first, 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 \((\text{last} - \text{first}) \times \log(\text{middle} - \text{first})\) 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 util::projection_identity

Parameters

- **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.

Return  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, typename Proj>
void partial_sort(ExPolicy &&policy,
RandomIt first,
RandomIt middle,
Sent last,
Comp &&comp = Comp(),
Proj &&proj = Proj())
```

Places the first \(\text{middle} - \text{first}\) elements from the range \([\text{first}, \text{last})\) as sorted with respect to \(\text{comp}\) into the range \([\text{first}, \text{middle})\). The rest of the elements in the range \([\text{middle}, \text{last})\) 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.

Note  Complexity: Approximately \((\text{last} - \text{first}) \times \log(\text{middle} - \text{first})\) 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 util::projection_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 \texttt{comp} is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type \texttt{parallel_policy} or \texttt{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** The \texttt{partial_sort} algorithm returns a `\texttt{hpx::future<RandomIt>}` if the execution policy is of type \texttt{sequenced_task_policy} or \texttt{parallel_task_policy} and returns \texttt{RandomIt} otherwise. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

```cpp
template<
    typename Rng,
    typename Comp,
    typename Proj>
hpx::traits::range_iterator<Rng> _thpx::ranges::partial_sort(Rng && rng, hpx::traits::range_iterator< Rng >_t middle, Comp && comp = Comp(), Proj && proj = Proj())
```

Places the first middle - first elements from the range \([\texttt{first}, \texttt{last})\) as sorted with respect to \texttt{comp} into the range \([\texttt{first}, \texttt{middle})\). The rest of the elements in the range \([\texttt{middle}, \texttt{last})\) are placed in an unspecified order.

The assignments in the parallel \texttt{partial_sort} algorithm invoked without an execution policy object execute in sequential order in the calling thread.

**Note** Complexity: Approximately \((\texttt{last} - \texttt{first}) \times \log(\texttt{middle} - \texttt{first})\) 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 \texttt{util::projection_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 \texttt{comp} is invoked.

**Return** The \texttt{partial_sort} algorithm returns `\texttt{typename hpx::traits::range_iterator<Rng>::type}`. It returns \texttt{last}.

```cpp
template<
    typename ExPolicy,
    typename Rng,
    typename Comp,
    typename Proj>
util::detail::algorithm_result_t<ExPolicy, ... && policy, Rng && rng, hpx::traits::range_iterator< Rng >_t middle, Comp && comp = Comp(), Proj && proj = Proj())
```

Sorts the elements in the range \([\texttt{first}, \texttt{last})\) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object \texttt{comp} (defaults to using operator\(<\))

A sequence is sorted with respect to a comparator \texttt{comp} and a projection \texttt{proj} if for every iterator \texttt{i} pointing to the sequence and every non-negative integer \texttt{n} such that \texttt{i} + \texttt{n} is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

**Note** Complexity: \(O(N \log(N))\), where \(N = \text{std::distance(first, last)}\) comparisons.
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.

**Template Parameters**
- `ExPolicy`: The type of the execution 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 `util::projection_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.

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.

**Return** The partial_sort algorithm returns a `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns `typename hpx::traits::range_iterator<Rng>::type` otherwise. It returns `last`.

```cpp
namespace hpx
namespace ranges

Functions

template<
  typename InIter, typename Sent1,
  typename RandIter, typename Sent2,
  typename Comp = Comp(),
  typename Proj1 = Proj1(),
  typename Proj2 = Proj2()
>
partial_sort_copy_result<
  InIter, RandIter>
partial_sort_copy(InIter first, Sent1 last, RandIter d_first, Sent2 d_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 [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 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 \(N = \text{std::distance(first, last)}\) and \(D = \text{std::distance(d_first, d_last)}\) comparisons.

**Template Parameters**
- `InIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an input iterator.
- **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`.
- **Proj1**: The type of an optional projection function for the input range. This defaults to `util::projection_identity`.
- **Proj2**: The type of an optional projection function for the output range. This defaults to `util::projection_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.
- **d_first**: Refers to the beginning of the destination range.
- **d_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.

**Return**
The `partial_sort_copy` algorithm returns a returns `partial_sort_copy_result<InIter, RandIter>`. The algorithm returns `{last, result_first + N}`.
Sorts some of the elements in the range \([\text{first}, \text{last})\) in ascending order, storing the result in the range \([\text{d_first}, \text{d_last})\). At most \(\text{d_last} - \text{d_first}\) of the elements are placed sorted to the range \([\text{d_first} + \text{n}, \text{d_last})\) where \(\text{n}\) is the number of elements to sort (\(\text{n} = \min(\text{last} - \text{first}, \text{d_last} - \text{d_first})\)).

The application of function objects in parallel algorithm invoked with an execution policy object of type \textit{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: \(O(N\log(\min(D,N)))\), where \(N = \text{std::distance(first, last)}\) and \(D = \text{std::distance(d\_first, d\_last)}\) comparisons.

\textbf{Template Parameters}

- \textit{ExPolicy}: The type of the execution policy to 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.
- \textit{FwdIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \textit{Sent1}: The type of the source sentinel (deduced). This sentinel type must be a sentinel for \textit{FwdIter}.
- \textit{RandIter}: The type of the destination iterators used (deduced). This iterator type must meet the requirements of a random iterator.
- \textit{Sent2}: The type of the destination sentinel (deduced). This sentinel type must be a sentinel for \textit{RandIter}.
- \textit{Comp}: The type of the function/function object to use (deduced). \textit{Comp} defaults to \textit{detail::less}.
- \textit{Proj1}: The type of an optional projection function for the input range. This defaults to \textit{util::projection\_identity}.
- \textit{Proj2}: The type of an optional projection function for the output range. This defaults to \textit{util::projection\_identity}.

\textbf{Parameters}

- \textit{policy}: The execution policy to use for the scheduling of the iterations.
- \textit{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \textit{last}: Refers to the sentinel value denoting the end of the sequence of elements the algorithm will be applied to.
- \textit{d\_first}: Refers to the beginning of the destination range.
- \textit{d\_last}: Refers to the sentinel denoting the end of the destination range.
- \textit{comp}: \textit{comp} is a callable object. The return value of the INVOKE operation applied to an object of type \textit{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 \textit{comp} will not apply any non-constant function through the dereferenced iterator. This defaults to \textit{detail::less}.
- \textit{proj1}: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation before the actual predicate \textit{comp} is invoked.
- \textit{proj2}: Specifies the function (or function object) which will be invoked for each pair of elements as a projection operation after the actual predicate \textit{comp} is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type \textit{parallel\_policy} or \textit{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \textit{partial\_sort\_copy} algorithm returns a \texttt{hpx::future<partial\_sort\_copy\_result<FwdIter, RandIter>>>} if the execution policy is of type \textit{sequenced\_task\_policy} or \textit{parallel\_task\_policy} and returns \textit{partial\_sort\_copy\_result<FwdIter, RandIter>} otherwise. The algorithm returns \(\{\text{last}, \text{result\_first} + \text{n}\}\).

\template<\typename ExPolicy, \typename FwdIter, \typename Sent1, \typename RandIter, \typename Sent2, \typename Comp, \typename Proj1, \typename Proj2>

\textbf{2.8. API reference} 539
partial_sort_copy_result<
    hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng2>>
    partial_sort_copy(
        Rng1 &&rng1,
        Rng2 &&rng2,
        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 [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 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 \(N = \text{std::distance(first, last)}\) and \(D = \text{std::distance(d_first, d_last)}\) 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 util::projection_identity.
- \(Proj1\): The type of an optional projection function for the output range. This defaults to util::projection_identity.

Parameters
- \(rng1\): Refers to the source range.
- \(rng1\): 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.

Return  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 FwdIter, typename Sent1, typename RandIter, typename Sent2, typename
partial_sort_copy

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 sequence_policy execute in sequential order in the calling thread.

Note Complexity: O(Nlog(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.
- `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 util::projection_identity.
- `Proj2`: The type of an optional projection function for the output range. This defaults to util::projection_identity.

Parameters
- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the source range.
- `rng1`: 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.

The application of function objects in parallel algorithm invoked with an execution policy object
of type \textit{parallel\_policy} or \textit{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \textit{partial\_sort\_copy} algorithm returns a \texttt{hpx::future<partial\_sort\_copy\_result<range\_iterator\_t<Rng1>, range\_iterator\_t<Rng2>>>} if the execution policy is of type \textit{sequenced\_task\_policy} or \textit{parallel\_task\_policy} and returns \textit{partial\_sort\_copy\_result<range\_iterator\_t<Rng1>, range\_iterator\_t<Rng2>>>} otherwise. The algorithm returns \{last, result\_first + N\}.

\texttt{namespace hpx}

\textbf{Functions}

\texttt{template<typename FwdIter, typename Sent, typename Pred, typename Proj> subrange_t<FwdIter> partition\!(ExPolicy && policy, Sent first, Sent last, Pred && pred, Proj && proj)}

Reorders the elements in the range \{first, last\} in such a way that all elements for which the predicate \texttt{pred} returns true precede the elements for which the predicate \texttt{pred} returns false. Relative order of the elements is not preserved.

The assignments in the parallel \textit{partition} algorithm invoked without an execution policy object execute in sequential order in the calling thread.

\textbf{Note} Complexity: At most 2 * (last - first) swaps. Exactly last - first applications of the predicate and projection.

\textbf{Template Parameters}

- \texttt{FwdIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{Sent}: The type of the source sentinel (deduced). This sentinel type must be a sentinel for FwdIter.
- \texttt{Pred}: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \textit{partition} requires \texttt{Pred} to meet the requirements of \texttt{CopyConstructible}.
- \texttt{Proj}: The type of an optional projection function. This defaults to \texttt{util::projection\_identity}

\textbf{Parameters}

- \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \texttt{last}: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
- \texttt{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:

\begin{verbatim}
bool pred(const Type &a);
\end{verbatim}

The signature does not need to have \texttt{const\&}, but the function must not modify the objects passed to it. The type \texttt{Type} must be such that an object of type \texttt{InIter} can be dereferenced and then implicitly converted to \texttt{Type}.

- \texttt{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.
Return The `partition` algorithm returns a `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>
util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter>>::type partition(
    ExPolicy &&policy,
    FwdIter first,
    Sent last,
    Pred &&pred,
    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.

Note Complexity: At most $2 \ast (\text{last} - \text{first})$ 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 `util::projection_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.
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.

**Return** 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.

```cpp
template<typename Rng, typename Pred, typename Proj>
subrange_t<hpx::traits::range_iterator_t<Rng>> partition(Rng &&rng, Pred &&pred, 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 `util::projection_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 the range `rng`. This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```cpp
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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename Pred, typename 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.

**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 `util::projection_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:

```cpp
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.

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.

**Return** 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 sub-
range_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 BidirIter, typename Sent, typename F, typename Proj>
subrange_t<BidirIter> stable_partition (BidirIter first, Sent last, F &&f, Proj &&proj)
Permutates 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.
- 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 util::projection_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.
- 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:

```cpp
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.

Return 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 F, typename Proj>
`util::detail::algorithm_result<ExPolicy, subrange_t<BidirIter>>`::type `stable_partition` (ExPolicy &&policy, BidirIter first, Sent last, F &&f, 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.

**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`.
- `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 `util::projection_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.
- `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:

```cpp
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.

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.
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 F, typename Proj>
subrange_t<hpx::traits::range_iterator_t<Rng>> stable_partition(Rng &&rng, F &&f, 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 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 `util::projection_identity`.

**Parameters**
- `rng`: Refers to 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:

  ```cpp
def 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.

**Return** 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.
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) \( \text{INVOKED}(f, \text{INVOKED}(\text{proj}, *j)) \neq \text{false} \), and for every iterator k in the range [i, last), \( \text{INVOKED}(f, \text{INVOKED}(\text{proj}, *k)) = \text{false} \)

The invocations of \( f \) in the parallel \textit{stable_partition} algorithm invoked with an execution policy object of type \textit{sequenced_policy} executes in sequential order in the calling thread.

\textbf{Note} Complexity: At most \((\text{last} - \text{first}) \times \log(\text{last} - \text{first})\) swaps, but only linear number of swaps if there is enough extra memory. Exactly \((\text{last} - \text{first})\) applications of the predicate and projection.

\textbf{Template Parameters}

- \textit{ExPolicy}: The type of the execution policy to 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 \( f \).
- \textit{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an \\textit{bidirectional iterator}.
- \textit{F}: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \textit{transform} requires \textit{F} to meet the requirements of \textit{CopyConstructible}.
- \textit{Proj}: The type of an optional projection function. This defaults to \textit{util::projection_identity}.

\textbf{Parameters}

- \textit{policy}: The execution policy to use for the scheduling of the iterations.
- \textit{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \textit{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 \([\text{first}, \text{last})\). The signature of this predicate should be equivalent to:

```cpp
bool fun(const Type &a);
```

The signature does not need to have \texttt{const\&}. The type \textit{Type} must be such that an object of type \textit{BidirIter} can be dereferenced and then implicitly converted to \textit{Type}.
- \textit{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.

The invocations of \( f \) in the parallel \textit{stable_partition} algorithm invoked with an execution policy object of type \textit{parallel_policy} or \textit{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \textit{stable_partition} algorithm returns an iterator \( i \) such that for every iterator \( j \) in the range \([\text{first}, \text{last})\), \( f(*j) \neq \text{false} \) \( \text{INVOKED}(f, \text{INVOKED}(\text{proj}, *j)) \neq \text{false} \), and for every iterator \( k \) in the range \([i, \text{last})\), \( f(*k) = \text{false} \) \( \text{INVOKED}(f, \text{INVOKED}(\text{proj}, *k)) = \text{false} \). 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.

```cpp
template<typename FwdIter1, typename Sent, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
hpx::util::tagged_tuple<tag::in(FwdIter1), tag::out1 FwdIter2, tag::out2 FwdIter3> partition_copy(FwdIter1 first, Sent last, FwdIter2 dest_true, FwdIter3 dest_false, Pred &&pred, 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**

- **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 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 `util::projection_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:

```cpp
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`. 

---

Chapter 2. What’s so special about **HPX**?
• 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.

Return 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
util::detail::algorithm_result<ExPolicy, partition_copy_result<FwdIter1, OutIter2, OutIter3>>::type partition_copy(
    ExPolicy&& policy,
    FwdIter1 first,
    Sent last,
    FwdIter2 dest_true,
    FwdIter3 dest_false,
    Pred&& pred,
    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.

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.
- 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 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`. 

2.8. API reference 551
• **Proj**: The type of an optional projection function. This defaults to `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
template<typename Rng, typename FwdIter2, typename FwdIter3, typename Pred, typename Proj>
friend partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>::type partition_copy
(Rng &&rng, FwdIter2 &&first, FwdIter3 &&last, Pred &&pred, 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 
\[ N = \text{std::distance}(\text{begin(rng)}, \text{end(rng)}). \]

**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.

- **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 `util::projection_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:

```cpp
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.

**Return** 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.
friend parallel::util::detail::algorithm_result<ExPolicy, partition_copy_result<hpx::traits::range_iterator_t<Rng>, FwdIter2, FwdIter3>> partition_copy(
  ExPolicy&& policy,
  Rng&& rng,
  FwdIter2 dest_true,
  FwdIter3 dest_false,
  Pred&& pred,
  Proj&& proj)

Copies the elements in the range \textit{rng}, to two different ranges depending on the value returned by the predicate \textit{pred}. The elements, that satisfy the predicate \textit{pred} are copied to the range beginning at \textit{dest_true}. The rest of the elements are copied to the range beginning at \textit{dest_false}. The order of the elements is preserved.

The assignments in the parallel \textit{partition_copy} algorithm invoked with an execution policy object of type \textit{sequenced_policy} execute in sequential order in the calling thread.

\textbf{Note}  Complexity: Performs not more than \(N\) assignments, exactly \(N\) applications of the predicate \textit{pred}, where \(N = \text{std::distance}(	ext{begin(rng)}, \text{end(rng)})\).

\textbf{Template Parameters}

- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- \texttt{FwdIter2}: The type of the iterator representing the destination range for the elements that satisfy the predicate \textit{pred} (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{FwdIter3}: The type of the iterator representing the destination range for the elements that don’t satisfy the predicate \textit{pred} (deduced). This iterator type must meet the requirements of an forward iterator.
- \texttt{Pred}: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \textit{partition_copy} requires \textit{Pred} to meet the requirements of \textit{CopyConstructible}.
- \texttt{Proj}: The type of an optional projection function. This defaults to \textit{util::projection_identity}.

\textbf{Parameters}

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \texttt{dest_true}: Refers to the beginning of the destination range for the elements that satisfy the predicate \textit{pred}.
- \texttt{dest_false}: Refers to the beginning of the destination range for the elements that don’t satisfy the predicate \textit{pred}.
• **pred**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```cpp
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.

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.

**Return** The `partition_copy` algorithm returns a `hpx::future<partition_copy_result<\text{range_iterator_t}_1, FwdIter2, FwdIter3>>` if the execution policy is of type `parallel_task_policy` and returns `partition_copy_result<\text{range_iterator_t}_1, 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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename ExPolicy, typename FwdIter, typename Sent, typename T, typename F>
util::detail::algorithm_result<ExPolicy, T>::type 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.

**Note** Complexity: \(O(\text{last} - \text{first})\) 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 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.

---

2.8. API reference 555
• 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:

```cpp
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.

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 un-
specified 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.

**Return** The reduce algorithm returns a hpx::future<T> if the execution policy is of type se-

```cpp
template<typename ExPolicy, typename FwdIter, typename Sent, typename T>
util::detail::algorithm_result<ExPolicy, T>::type reduce(ExPolicy &&policy, FwdIter first, Sent last, T init)
```

Returns GENERALIZED_SUM(+, init, *(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.

**Note** Complexity: O(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 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 re-
quirements 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.

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 un-
specified 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.

**Return** 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)`.

**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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Sent>
util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<FwdIter>::value_type>::type
reduce(ExPolicy &&policy, FwdIter first, Sent last)
```

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.

**Note** Complexity: O(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 begin iterator used (deduced). This iterator type must meet the requirements of a 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.

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.

**Return** 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)`.

**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:
- \( b_1, \ldots, b_N \) may be any permutation of \( a_1, \ldots, a_N \) and
- \( 1 < K+1 = M \leq N \).

```
template<typename ExPolicy, typename Rng, typename T, typename F>
util::detail::algorithm_result<ExPolicy, T>::type reduce
(ExPolicy &&policy, Rng &&rng, T init, F &&f)
Returns GENERALIZED_SUM(f, init, *first, \ldots, *(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.

**Note** Complexity: \( O(last - first) \) applications of the predicate \( f \).

**Template Parameters**
- \( \text{ExPolicy} \): The type of the execution policy to 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.
- \( \text{Rng} \): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \( \text{F} \): The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \( \text{copy_if} \) requires \( \text{F} \) to meet the requirements of \( \text{CopyConstructible} \).
- \( \text{T} \): The type of the value to be used as initial (and intermediate) values (deduced).

**Parameters**
- \( \text{policy} \): The execution policy to use for the scheduling of the iterations.
- \( \text{rng} \): Refers to the sequence of elements the algorithm will be applied to.
- \( \text{f} \): Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a binary predicate. The signature of this predicate should be equivalent to:

```c
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&. The types \( \text{Type1} \ Ret \) must be such that an object of type \( \text{FwdIterB} \) can be dereferenced and then implicitly converted to any of those types.
- \( \text{init} \): The initial value for the generalized sum.

The reduce operations in the parallel \( \text{copy_if} \) 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.

**Return** The reduce algorithm returns a \( \text{hpx::future<T>} \) 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 \([\text{first}, \text{last})\).

**Note** GENERALIZED_SUM(op, a_1, \ldots, a_N) is defined as follows:
- \( a_1 \) when \( N = 1 \)
- \( \text{op} (\text{GENERALIZED_SUM}(\text{op}, b_1, \ldots, b_K), \text{GENERALIZED_SUM}(\text{op}, b_M, \ldots, b_N)) \),
  where:
  - \( b_1, \ldots, b_N \) may be any permutation of \( a_1, \ldots, a_N \) and
  - \( 1 < K+1 = M \leq N \).

```
template<typename ExPolicy, typename Rng, typename T>
util::detail::algorithm_result<ExPolicy, T>::type reduce
(ExPolicy &&policy, Rng &&rng, T init)
Returns GENERALIZED_SUM(+, init, *first, \ldots, *(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.

**Note** Complexity: \(O(\text{last} - \text{first})\) 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 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.

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.

**Return** 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 \([\text{first}, \text{last})\).

**Note** GENERALIZED_SUM(+, a1, \ldots, aN) is defined as follows:
- \(a1\) when \(N\) is 1
- \(\text{op}(\text{GENERALIZED_SUM}(+, b1, \ldots, bK), \text{GENERALIZED_SUM}(+, bM, \ldots, bN))\), where:
  - \(b1, \ldots, bN\) may be any permutation of \(a1, \ldots, aN\) and
  - \(1 < K+1 = M \leq N\).

\[
\begin{align*}
\text{template<typename ExPolicy, typename Rng> return util::detail::algorithm_result<ExPolicy, typename std::iterator_traits<typename hpx::traits::range_traits<Rng>::iterator_type>::value_type>::type reduce(ExPolicy&& policy, Rng&& rng)}
\end{align*}
\]

Returns GENERALIZED_SUM(+, T(), *first, \ldots, *(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.

**Note** Complexity: \(O(\text{last} - \text{first})\) 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.

**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.

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 un-
specified 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.

**Return** 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)`.

**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`.

namespace hpx

namespace ranges

**Functions**

```cpp
template<typename FwdIter, typename Sent, typename T, typename Proj = util::projection_identity>
subrange_t<FwdIter, Sent> remove(FwdIter first, Sent last, T const &value, Proj &&proj = Proj())
```

Removes all elements that are equal to `value` 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` 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**
- `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 `util::projection_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.

**Return** 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.
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.

**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 algorithm. 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 `util::projection_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.

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.

**Return** 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.

```cpp
template<typename Rng, typename T, typename Proj = util::projection_identity>
subrange_t<hpx::traits::range_iterator<Rng>::type> 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)` 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 `util::projection_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.

**Return** The remove algorithm returns a subrange `[ret, nutil::end(rng))`, where `ret` is a past-the-end iterator for the new subrange of the values all in valid but unspecified state.

```
template<typename ExPolicy, typename Rng, typename T, typename Proj = util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<typename hpx::traits::range_iterator<Rng>::type>> 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.

**Note** Complexity: Performs not more than `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 `util::projection_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.

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.

**Return** The remove algorithm returns a `hpx::future<> subrange_t<typename 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 remove algorithm returns the iterator to the new end of the range.

```cpp
template<typename FwdIter, typename Sent, typename Pred = hpx::parallel::util::projection_identity>
subrange_t<FwdIter, Sent> remove_if (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 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` and the projection `proj`.

**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 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 `util::projection_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 which returns `true` for the required elements. The signature of this predicate should be equivalent to:

```cpp
def 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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Sent, typename Pred, typename Proj = hpx::parallel::util::projection_identity>
```
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.

**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 algorithm. 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 `util::projection_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 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.

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.

**Return** 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 Rng, 
typename T, 
typename Proj = util::projection_identity>
subrange_t<typename 
hpx::traits::range_iterator<Rng>::type> 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 oper-
ator==( ) 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 CopyCon-
structible.
•  Proj: The type of an optional projection function. This defaults to util::projection_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 ele-
ments 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:

```cpp
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 ele-
ments as a projection operation before the actual predicate is invoked.

Return The remove_if algorithm returns a subrange_t<
typename 
hpx::traits::range_iterator<Rng>::type>. 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.

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.

**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 `util::projection_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:

```cpp
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.

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.

**Return** The `remove_if` algorithm returns a `hpx::future<subrange_t<typename 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 `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.

```cpp
namespace hpx
{
    namespace ranges
    {

        Functions

        template<typename Iter, typename Sent, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_identity>
        Iter replace(Iter first, Sent sent, T1 const &old_value, T2 const &new_value, Proj &&proj = Proj())
        {
            // Implementation...
        }
    }
}
```

**Effects**: Substitutes elements referred by the iterator it in the range `[first,last)` with `new_value`, when the following corresponding conditions hold: `INVOKEd(proj, *i) == old_value`
Note Complexity: Performs exactly last - first assignments.

The assignments in the parallel replace algorithm execute in sequential order in the calling thread.

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 util::projection_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.
- 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.

Return The replace algorithm returns an Iter.

template<typename Rng, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_identity>
   hpx::traits::range_iterator<Rng>::type
   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

Note Complexity: Performs exactly util::end(rng) - util::begin(rng) assignments.

The assignments in the parallel replace algorithm execute in sequential order in the calling thread.

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 util::projection_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.

Return The replace algorithm returns an hpx::traits::range_iterator<Rng>::type.

template<typename ExPolicy, typename Iter, typename Sent, typename T1, typename T2, typename Proj = hpx::parallel::util::detail::algorithm_result<ExPolicy, Iter>::type>
   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 \([\text{first}, \text{last})\) with new_value, when the following corresponding conditions hold: \(\text{INVOKE}(\text{proj}, *i) == \text{old_value}\)

**Note** Complexity: Performs exactly \(\text{last} - \text{first}\) assignments.

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.

**Template Parameters**
- `ExPolicy`: The type of the execution policy to 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 `util::projection_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.
- `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 `parallel_policy` or `parallel_task_policy` are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename T1, typename T2, typename Proj = util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type
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`

**Note** Complexity: Performs exactly `util::end(rng) - util::begin(rng)` assignments.

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.

**Template Parameters**
- `ExPolicy`: The type of the execution policy to 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 `util::projection_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.

**Return** 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.

```cpp
template<typename Iter, typename Sent, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_identity>
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`

**Note** Complexity: Performs exactly `sent - first` applications of the predicate.

The assignments in the parallel replace_if algorithm execute in sequential order in the calling 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.
- `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 `util::projection_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 which returns `true` for the elements which need to replaced. The signature of this predicate should be equivalent to:

```cpp
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.

**Return** The replace_if algorithm returns an `Iter` it returns last.

```cpp
template<
typename Rng, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_identity>
  hpx::traits::range_iterator<Rng>::type
  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.

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.

**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 `util::projection_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:

```cpp
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.
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.

Return 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 T, typename Proj = hpx::parallel::util::projection_identity>
```

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.

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.

Template Parameters
- ExPolicy: The type of the execution policy to 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 util::projection_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 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.

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.

Return 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 T, typename Proj = hpx::parallel::util::projection_identity>
```
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.

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.

**Template Parameters**

- `ExPolicy`: The type of the execution policy to 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 `util::projection_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:

```cpp
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.

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.
**Return** The `replace_if` algorithm returns a `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy`. It returns `last`.

```cpp
template<typename InIter, typename Sent, typename OutIter, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_identity>
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**

- **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.
- **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 `util::projection_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.

**Return** 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.

```cpp
template<typename Rng, typename OutIter, typename T1, typename T2, typename Proj = hpx::parallel::util::projection_identity>
```
replace_copy_result<
typename hpx::traits::range_iterator<Rng>::type, 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 + (i - 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 util::projection_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.

Return The replace_copy algorithm returns an in_out_result<typename hpx::traits::range_iterator<Rng>::type, 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 T1, typename T2>
`parallel::util::detail::algorithm_result<ExPolicy, replace_copy_result<FwdIter1, FwdIter2>>::type replace_copy (ExPolicy&& policy, FwdIter1 first, Sent sent, FwdIter2 dest, T1 old_value, T2 new_value, Proj&& proj = Proj())`

Copies all the 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.

**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 `util::projection_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.

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.
Return 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.

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.

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 `util::projection_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.
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.

Return The replace_copy algorithm returns a hpx::future<in_out_result<
  typename hpx::traits::range_iterator<Rng>::type, FwdIter>> if the execution policy is of type sequenced_task_policy or parallel_task_policy and returns in_out_result<
  typename hpx::traits::range_iterator<Rng>::type, 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.

template<typename InIter, typename Sent, typename OutIter, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_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.
- T: The type of the new values to replace (deduced).
- Proj: The type of an optional projection function. This defaults to util::projection_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.
Return The replace_copy_if algorithm returns a \(\text{in\_out\_result<InIter, OutIter>}\). The replace_copy_if algorithm returns the input iterator \(\text{last}\) and the output iterator to the element in the destination range, one past the last element copied.

```cpp
template<typename Rng, typename OutIter, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_identity>
replace_copy_if_result<typename hpx::traits::range_iterator<Rng>::type, OutIter> replace_copy_if(Rng &&rng, OutIter dest, Pred &&pred, T const new_value, Proj &&proj = Proj())
```

Copies the all elements from the range \(\text{rng}\) to another range beginning at \(\text{dest}\) replacing all elements satisfying a specific criteria with \(\text{new\_value}\).

Effects: Assigns to every iterator \(\text{it}\) in the range \([\text{result}, \text{result} + (\text{util::end(rng)} - \text{util::begin(rng)})\)) either \(\text{new\_value}\) or \(*\text{first} + (\text{it} - \text{result})\) depending on whether the following corresponding condition holds: INVOKE(f, INVOKE(proj, *\(\text{first} + (\text{it} - \text{result})\))) != false

The assignments in the parallel replace_copy_if algorithm execute in sequential order in the calling thread.

Note Complexity: Performs exactly \(\text{util::end(rng)} - \text{util::begin(rng)}\) applications of the predicate.

Template Parameters
- \(\text{Rng}\): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \(\text{OutIter}\): The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- \(\text{Pred}\): The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \(\text{equal}\) requires \(\text{Pred}\) to meet the requirements of \(\text{CopyConstructible}\).
- \(\text{T}\): The type of the new values to replace (deduced).
- \(\text{Proj}\): The type of an optional projection function. This defaults to \(\text{util::projection_identity}\)

Parameters
- \(\text{rng}\): Refers to the sequence of elements the algorithm will be applied to.
- \(\text{dest}\): Refers to the beginning of the destination range.
- \(\text{pred}\): Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is an unary predicate which returns \(\text{true}\) for the elements which need to replaced. The signature of this predicate should be equivalent to:

```cpp
bool pred(const Type &a);
```

The signature does not need to have \(\text{const}\&\), but the function must not modify the objects passed to it. The type \(\text{Type}\) must be such that an object of type \(\text{FwdIter}\) can be dereferenced and then implicitly converted to \(\text{Type}\).
- \(\text{new\_value}\): 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.

**Return** The `replace_copy_if` algorithm returns an `in_out_result<typename hpx::traits::range_iterator<Rng>::type, 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 Proj = hpx::parallel::util::projection_identity>

parallel::util::detail::algorithm_result<ExPolicy, replace_copy_if_result<FwdIter1, FwdIter2>>::type

replace_copy_if(ExPolicy&& policy, FwdIter1 first, Sent sent, FwdIter2 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 invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling 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 `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename FwdIter, typename Pred, typename T, typename Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, replace_copy_if_result<typename hpx::traits::range_iterator<Rng>::type, FwdIter>>::type replace_copy_if(ExPolicy&& policy, Rng&& rng, FwdIter 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 invoked with an execution policy object of type sequenced_policy execute in sequential order in the calling 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.
• **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 `util::projection_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:

```cpp
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.

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.

**Return** The `replace_copy_if` algorithm returns an `hpx::future<in_out_result<typename hpx::traits::range_iterator<Rng>::type, 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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename Iter, typename Sent>
Iter reverse (Iter first, Sent 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**

• **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`. 581
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.

**Return** The reverse algorithm returns a Iter. It returns last.

```cpp
template<typename Rng>
hpx::traits::range_iterator<Rng>::type 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.

**Return** The reverse algorithm returns a `hpx::traits::range_iterator<Rng>::type`. It returns last.

```cpp
template<typename ExPolicy, typename Iter, typename Sent>
parallel::util::detail::algorithm_result<ExPolicy, Iter>::type reverse (ExPolicy &&policy, Iter first, Sent 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 `sequence_policy` execute in sequential order in the calling 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.
- **last**: Refers to the end of the sequence of elements the algorithm will be applied to.

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.

**Return** The reverse algorithm returns a `hpx::future<Iter>` if the execution policy is of type `sequence_policy` or `parallel_task_policy` and returns Iter otherwise. It returns last.
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.

**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.

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.

**Return** The `reverse` algorithm returns a `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` otherwise. It returns `last`.

```cpp
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.
Return 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.

```cpp
template<
    typename Rng, typename OutIter>
ranges::reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, 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.
- `OutputIter`: 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.

**Return** The `reverse_copy` algorithm returns a `ranges::reverse_copy_result<` type

```cpp
template<
    typename ExPolicy, typename Iter, typename Sent, typename OutIter>
parallel::util::detail::algorithm_result<ExPolicy, reverse_copy_result<Iter, OutIter>> reverse_copy(ExPolicy &&policy, 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 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**
- **ExPolicy**: The type of the execution policy to 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.
- **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.
- **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.

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.

**Return** The reverse_copy algorithm returns a hpx::future<hpx::traits::range_iterator<Rng>::type, OutIter>> if the execution policy is of type sequenced_task_policy or parallel_task_policy and returns reverse_copy_result<Iter, OutIter> 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.

```cpp
template<typename ExPolicy, typename Rng, typename OutIter>
util::detail::algorithm_result<ExPolicy, ranges::reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>>::type
reverse_copy(ExPolicy&& policy, 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 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**
- **ExPolicy**: The type of the execution policy to 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.

- **OutputIter**: 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.

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.

**Return** The `reverse_copy` algorithm returns a `hpx::future<ranges::reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `ranges::reverse_copy_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>` otherwise. The `reverse_copy` algorithm returns an object equal to `{last, result + N}` where `N = last - first`.

```cpp
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`.

**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.

**Note** The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable` and `Move-Constructible`.

**Return** The `rotate` algorithm returns a `subrange_t<FwdIter, Sent>`. The `rotate` algorithm returns the iterator equal to pair(`first + (last - middle)`, `last`).

```cpp
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 [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.

**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 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.

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** The type of dereferenced FwdIter must meet the requirements of MoveAssignable and MoveConstructible.

**Return** 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<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`.

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.

Note The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return 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>
util::detail::algorithm_result<ExPolicy, subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> rotate(ExPolicy&& policy, Rng&& rng, hpx::traits::range_iterator_t<Rng> 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 invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling 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 forward iterator.

Parameters
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `policy`: The execution policy to use for the scheduling of the iterations.
- `middle`: Refers to the element that should appear at the beginning of the rotated range.

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 The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return 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.

Return 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.

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.
• **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.

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.

**Return** The `rotate_copy` algorithm returns are turned `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.

```cpp
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**: Refers to the begin of the destination range.

**Return** 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.

```cpp

```

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.

**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: Refers to the begin of the destination range.

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.

**Return** The rotate_copy algorithm returns a hpx::future<rotate_copy_result<hpx::traits::range_iterator_t<Rng>, OutIter>> if the execution policy is of type parallel_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.

```cpp
namespace hpx

namespace ranges

Functions

template<
typename FwdIter, typename Sent, typename FwdIter2, typename Sent2, typename Pred = hpx::ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_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 a 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 `util::projection_identity` and is applied to the elements of type dereferenced `FwdIter`.

• **Proj2**: The type of an optional projection function. This defaults to `util::projection_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.

• **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 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 should be equivalent to

  ```cpp
  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.

**Return**

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.

```cpp
template<typename ExPolicy, typename FwdIter, typename Sent, typename FwdIter2, typename Sent2, typename Pred = hpx::ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
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.

**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 `util::projection_identity` and is applied to the elements of type dereferenced `FwdIter`.
- **Proj2**: The type of an optional projection function. This defaults to `util::projection_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

```cpp
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.

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.

**Return** 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.

```cpp
template<typename Rng1, typename Rng2, typename Pred = hpx::ranges::equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
```
hpx::traits::range_iterator<Rng1>::type 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 util::projection_identity and is applied to the elements of Rng1.
- **Proj2**: The type of an optional projection function. This defaults to util::projection_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

```cpp
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.

**Return** 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.
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 \texttt{search} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: at most \((S\times N)\) comparisons where \(S = \text{distance}(s\_first, s\_last)\) and \(N = \text{distance}(\text{first, last})\).

\textbf{Template Parameters}

- \texttt{ExPolicy}: The type of the execution policy to 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.
- \texttt{Rng1}: The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \texttt{Rng2}: The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \texttt{Pred}: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of \texttt{adjacent\_find} requires \texttt{Pred} to meet the requirements of \texttt{CopyConstructible}. This defaults to \texttt{std::equal\_to}.
- \texttt{Proj1}: The type of an optional projection function. This defaults to \texttt{util::projection\_identity} and is applied to the elements of \texttt{Rng1}.
- \texttt{Proj2}: The type of an optional projection function. This defaults to \texttt{util::projection\_identity} and is applied to the elements of \texttt{Rng2}.

\textbf{Parameters}

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{rng1}: Refers to the sequence of elements the algorithm will be examining.
- \texttt{rng2}: Refers to the sequence of elements the algorithm will be searching for.
- \texttt{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

\begin{verbatim}
bool pred(const Type1 &a, const Type2 &b);
\end{verbatim}

The signature does not need to have \texttt{const &}, but the function must not modify the objects passed to it. The types \texttt{Type1} and \texttt{Type2} must be such that objects of types \texttt{FwdIter1} and \texttt{FwdIter2} can be dereferenced and then implicitly converted to \texttt{Type1} and \texttt{Type2} respectively.
• **proj1**: Specifies the function (or function object) which will be invoked for each of the elements of \( rmg1 \) 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 \( rmg2 \) as a projection operation before the actual predicate is invoked.

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.

**Return** 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.

```cpp
template<typename FwdIter, typename FwdIter2, typename Sent2, typename Pred = hpx::ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
FwdIter search_n(ExPolicy &&policy, 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 \(S = \text{distance}(s\_first, s\_last)\) and \(N = \text{count}\).

**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<>`

**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

```cpp
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.

---

596 Chapter 2. What’s so special about HPX?
• **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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter, typename FwdIter2, typename Sent2, typename Pred = hpx::ranges::equal_to, typename Proj1 = parallel::util::projection_identity, typename Proj2 = parallel::util::projection_identity>
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.

**Note** Complexity: at most `(S*N)` comparisons where \( S = \text{distance}(s\_first, s\_last) \) and \( N = \text{count} \).

**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<>`

**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

  ```cpp
  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.

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.

**Return** 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.

```cpp
template<typename Rng1, typename Rng2, typename Pred = hpx::ranges::equal_to, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>

hpx::traits::range_iterator<Rng1>::type
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)` using 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 `util::projection_identity` and is applied to the elements of `Rng1`.
- **Proj2**: The type of an optional projection function. This defaults to `util::projection_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

```cpp
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.

---

598 Chapter 2. What’s so special about HPX?
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.

Return 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 Parameters
- ExPolicy: The type of the execution policy to 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 util::projection_identity and is applied to the elements of Rng1.
- Proj2: The type of an optional projection function. This defaults to util::projection_identity.

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.

Note Complexity: at most (S*N) comparisons where S = distance(s_first, s_last) and N = distance(first, last).
and is applied to the elements of \textit{Rng2}.

\textbf{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

\begin{verbatim}
bool pred(const Type1 &a, const Type2 &b);
\end{verbatim}

The signature does not need to have const &, but the function must not modify the objects passed to it. The types \textit{Type1} and \textit{Type2} must be such that objects of types \textit{FwdIter1} and \textit{FwdIter2} can be dereferenced and then implicitly converted to \textit{Type1} and \textit{Type2} respectively.

- **proj1**: Specifies the function (or function object) which will be invoked for each of the elements of \textit{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 \textit{rng2} as a projection operation before the actual predicate is invoked.

The comparison operations in the parallel search algorithm invoked with an execution policy object of type \textit{parallel\_policy} or \textit{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The search algorithm returns a \textit{hpx::future<FwdIter>} if the execution policy is of type \textit{task\_execution\_policy} and returns \textit{FwdIter} otherwise. The search algorithm returns an iterator to the beginning of the first subsequence \([s\text{\_first}, s\text{\_last})\) in range \([first, last)\). If the length of the subsequence \([s\text{\_first}, s\text{\_last})\) is greater than the length of the range \([first, last)\), \textit{last} is returned. Additionally if the size of the subsequence is empty \textit{first} is returned. If no subsequence is found, \textit{last} is returned.

\begin{verbatim}
namespace hpx

namespace ranges

\end{verbatim}

\textbf{Functions}

\begin{verbatim}
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Pred = detail::less, typename Proj1 = util::projection_identity, typename Proj2 = util::projection_identity>

\end{verbatim}
util::detail::algorithm_result<ExPolicy, ranges::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 \( \text{std::max}(m-n, 0) \) times. The resulting range cannot overlap with either of the input ranges.

Note Complexity: At most \( 2 \times (N_1 + N_2 - 1) \) comparisons, where \( N_1 \) is the length of the first sequence and \( N_2 \) is the length of the second sequence.

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 (sequence_policy) or in a single new thread spawned from the current thread (for sequence_task_policy).

**Template Parameters**

- **ExPolicy**: The type of the execution 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) representing the first sequence. 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
• `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 `util::projection_identity`.

• `Proj2`: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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:

```cpp
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.

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.

**Return**

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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Iter3, typename Pred = detail::less, typename...
```
util::detail::algorithm_result<ExPolicy, ranges::set_difference_result<typename traits::range_iterator<Rng1>::type, Iter3>::type> set_difference(ExPolicy&& policy, 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.

**Note** Complexity: At most \( 2 \times (N_1 + N_2 - 1) \) comparisons, where \( N_1 \) is the length of the first sequence and \( N_2 \) is the length of the second sequence.

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).

**Template Parameters**

- **ExPolicy**: The type of the execution 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 `util::projection_identity`
- **Proj2**: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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:

```cpp
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.

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.

**Return**

The *set_difference* algorithm returns a *hpx::future<ranges::set_difference_result<Iter1, Iter3>>* if the execution policy is of type *sequenced_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.

```cpp
namespace hpx

namespace ranges

Functions
```

```cpp
template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3>
```
util::detail::algorithm_result<ExPolicy, ranges::set_intersection_result<Iter1, Iter2, Iter3>::type set_intersection

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 \( \min(m, n) \) 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.

**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.

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 (\( \text{sequenced\_policy} \)) or in a single new thread spawned from the current thread (for \( \text{sequenced\_task\_policy} \)).

**Template Parameters**
- **ExPolicy**: The type of the execution 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 `util::projection_identity`
- **Proj2**: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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:

```cpp
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.

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.

### Return

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.
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.

**Note** Complexity: At most \( 2^*(N_1 + N_2 - 1) \) comparisons, where \( N_1 \) is the length of the first sequence and \( N_2 \) is the length of the second sequence.

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).

**Template Parameters**
- \( \text{ExPolicy} \): The type of the execution 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.
- \( \text{Rng1} \): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \( \text{Rng2} \): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \( \text{Iter3} \): The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- \( \text{Pred} \): The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of set_intersection requires \( \text{Pred} \) to meet the requirements of CopyConstructible. This defaults to std::less<>
- \( \text{Proj1} \): The type of an optional projection function applied to the first sequence. This defaults to util::projection_identity
- \( \text{Proj2} \): The type of an optional projection function applied to the second sequence. This defaults to util::projection_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:

```cpp
template<typename Type1> 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.

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.

**Return** 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.

```cpp
namespace hpx

namespace ranges

Functions

template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3>
```
Constructs a sorted range beginning at dest consisting of all elements present in either of the sorted ranges \([\text{first}_1, \text{last}_1)\) and \([\text{first}_2, \text{last}_2)\), but not in both of them are copied to the range beginning at \(\text{dest}\). 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 \([\text{first}_1, \text{last}_1)\) and \(n\) times in \([\text{first}_2, \text{last}_2)\), it will be copied to \(\text{dest}\) exactly \(\text{std::abs}(m-n)\) times. If \(m>n\), then the last \(m-n\) of those elements are copied from \([\text{first}_1,\text{last}_1)\), otherwise the last \(n-m\) elements are copied from \([\text{first}_2,\text{last}_2)\). The resulting range cannot overlap with either of the input ranges.

**Note** Complexity: At most \(2*(N_1 + N_2 - 1)\) comparisons, where \(N_1\) is the length of the first sequence and \(N_2\) is the length of the second sequence.

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 (\(\text{sequenced\_policy}\)) or in a single new thread spawned from the current thread (for \(\text{sequenced\_task\_policy}\)).

**Template Parameters**

- **ExPolicy**: The type of the execution 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 `util::projection_identity`
- **Proj2**: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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:

```cpp
def 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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Iter3, typename Pred = detail::less, typename
```
util::detail::algorithm_result<ExPolicy, ranges::set_symmetric_difference_result<typename traits::range_iterator<Rng1>::type, typename traits::range_iterator<Rng2>::type, Iter3>>::type set_symmetric_difference(ExPolicy&& policy, 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 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.

**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. 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).

**Template Parameters**
- **ExPolicy**: The type of the execution 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 util::projection_identity
• Proj2: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
namespace hpx
{
    namespace ranges
    {
        template<typename ExPolicy, typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3
```
util::detail::algorithm_result<ExPolicy, ranges::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.

**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.

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 (sequence_policy) or in a single new thread spawned from the current thread (for sequenced_task_policy).

**Template Parameters**
- ExPolicy: The type of the execution 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 `util::projection_identity`

• **Proj2**: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Iter3, typename Pred = detail::less, typename
```
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.

**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.

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`).

**Template Parameters**
- `ExPolicy`: The type of the execution 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 `util::projection_identity`
- `Proj2`: The type of an optional projection function applied to the second sequence. This defaults to `util::projection_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.

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.

Return

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.

```
namespace hpx

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.

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.

• **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.

**Note** The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

**Return** The `shift_left` algorithm returns `FwdIter`. The `shift_left` algorithm returns an iterator to the end of the resulting range.

```cpp
template<typename ExPolicy, typename FwdIter, typename Sent, typename Size>
parallel::util::detail::algorithm_result<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.

**Note** Complexity: At most `(last - first) - n` 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 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.

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** The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`.

**Return** 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 \([0, \text{last} - \text{first})\), 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.

**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.

**Note** The type of dereferenced `hpx::traits::range_iterator_t<Rng>` must meet the requirements of `MoveAssignable`.

**Return** 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>>::type 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 \([0, \text{last} - \text{first})\), 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.

**Note** Complexity: At most (last - first) - n 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 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.

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** The type of dereferenced `hpx::traits::range_iterator_t<Rng>` must meet the requirements of `MoveAssignable`.

**Return** 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.

namespace hpx

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.

**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.
- **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.

**Note** The type of dereferenced `FwdIter` must meet the requirements of `MoveAssignable`. 
The \textit{shift\_right} algorithm returns an iterator to the end of the resulting range.

\texttt{template<typename ExPolicy, typename FwdIter, typename Sent, typename Size> parallel::util::detail::algorithm_result<ExPolicy, FwdIter> shift\_right (ExPolicy&& policy, FwdIter first, Sent last, Size n)}

Shifts the elements in the range \([\text{first}, \text{last})\) by \(n\) positions towards the end of the range. For every integer \(i\) in \([0, \text{last} - \text{first} - n)\), moves the element originally at position \(\text{first} + i\) to position \(\text{first} \cdot n + i\).

The assignment operations in the parallel \textit{shift\_right} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: At most \((\text{last} - \text{first}) - n\) assignments.

\textbf{Template Parameters}

\begin{itemize}
\item \texttt{ExPolicy}: The type of the execution policy to 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.
\item \texttt{FwdIter}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
\item \texttt{Sent}: The type of the source sentinel (deduced). This sentinel type must be a sentinel for \texttt{FwdIter}.
\item \texttt{Size}: The type of the argument specifying the number of positions to shift by.
\end{itemize}

\textbf{Parameters}

\begin{itemize}
\item \texttt{policy}: The execution policy to use for the scheduling of the iterations.
\item \texttt{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
\item \texttt{last}: Refers to sentinel value denoting the end of the sequence of elements the algorithm will be applied.
\item \texttt{n}: Refers to the number of positions to shift.
\end{itemize}

The assignment operations in the parallel \textit{shift\_right} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Note} The type of dereferenced \texttt{FwdIter} must meet the requirements of \texttt{MoveAssignable}.

\textbf{Return} The \textit{shift\_right} algorithm returns a \texttt{hpx::future<FwdIter>} if the execution policy is of type \texttt{sequenced\_task\_policy} or \texttt{parallel\_task\_policy} and returns \texttt{FwdIter} otherwise. The \textit{shift\_right} algorithm returns an iterator to the end of the resulting range.

\texttt{template<typename Rng, typename Size> hpx::traits::range_iterator_t<Rng> shift\_right (Rng&& rng, Size n)}

Shifts the elements in the range \([\text{first}, \text{last})\) by \(n\) positions towards the end of the range. For every integer \(i\) in \([0, \text{last} - \text{first} - n)\), moves the element originally at position \(\text{first} + i\) to position \(\text{first} \cdot n + i\).
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 \((\text{last} - \text{first}) - n\) assignments.

**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.

**Note** The type of dereferenced `hpx::traits::range_iterator_t<Rng>` must meet the requirements of `Move-Assignable`.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, hpx::traits::range_iterator_t<Rng>>::type shift_right(ExPolicy&& policy, Rng&& rng, Size n)
```

Shifts the elements in the range \([\text{first}, \text{last})\) by \(n\) positions towards the end of the range. For every integer \(i\) in \([0, \text{last} - \text{first} - n)\), moves the element originally at position \(\text{first} + i\) to position \(\text{first} + n + i\).

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.

**Note** Complexity: At most \((\text{last} - \text{first}) - n\) 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 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.
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** The type of dereferenced `hpx::traits::range_iterator_t<Rng>` must meet the requirements of **Move-Assignable**.

**Return** 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.

```cpp
namespace hpx

namespace ranges

Functions

template<
  typename RandomIt,
  typename Sent,
  typename Comp,
  typename Proj
>
RandomIt sort(RandomIt first, Sent 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<()`).

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 \( \text{INVOKE}(\text{comp}, \text{INVOKE}(\text{proj}, *(i + n)), \text{INVOKE}(\text{proj}, *i)) == \text{false} \).

**Note** Complexity: \( O(N\log(N)) \), where \( N = \text{detail::distance}(\text{first}, \text{last}) \) comparisons.

`comp` has to induce a strict weak ordering on the values.

**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 `util::projection_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.

The assignments in the parallel `sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.
Return 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, typename Proj>

 parallel::util::detail::algorithm_result<ExPolicy, RandomIt>::type sort (ExPolicy &&policy, RandomIt first, Sent 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<()).

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.

Note Complexity: O(Nlog(N)), where N = detail::distance(first, last) comparisons. comp has to induce a strict weak ordering on the values.

Template Parameters
- ExPolicy: The type of the execution policy to 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 util::projection_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.

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.

Return 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<Rng>::type sort (Rng &&rng, Compare &&comp, 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.`

**Note** Complexity: $O(N\log(N))$, where $N = \text{std::distance(begin(rng), end(rng))}$ comparisons. `comp` has to induce a strict weak ordering on the values.

**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 `util::projection_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.

The assignments in the parallel `sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

**Return** The `sort` algorithm returns `typename hpx::traits::range_iterator<Rng>::type`. It returns `last`.

```cpp
template<typename ExPolicy, typename Rng, typename Pred, typename Proj>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type sort
(ExPolicy &&policy,
 Rng &&rng,
 Pred &&pred,
 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.`

**Note** Complexity: $O(N\log(N))$, where $N = \text{std::distance(begin(rng), end(rng))}$ comparisons. `comp` has to induce a strict weak ordering on the values.

**Template Parameters**
- `ExPolicy`: The type of the execution 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 `util::projection_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.

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.

Return  The sort algorithm returns a `hpx::future<typename hpx::traits::range_iterator<Rng>::type>` if the execution policy is of type `sequenced_task_policy` or `parallel_task_policy` and returns `typename hpx::traits::range_iterator<Rng>::type` otherwise. It returns last.

namespace hpx

namespace ranges

Functions

template<typename RandomIt, typename Sent, typename Comp, typename Proj>
RandomIt stable_sort (RandomIt first, Sent last, Comp &comp, 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.

Note  Complexity: $O(N\log(N))$, where $N = \text{std::distance}(\text{first}, \text{last})$ comparisons. `comp` has to induce a strict weak ordering on the values.

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 `util::projection_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.

The assignments in the parallel `stable_sort` algorithm invoked without an execution policy object execute in sequential order in the calling thread.

**Return**  The `stable_sort` algorithm returns `RandomIt`. The algorithm returns an iterator pointing to the first element after the last element in the input sequence.

```cpp
template<typename ExPolicy, typename RandomIt, typename Sent, typename Comp, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, RandomIt>::type stable_sort(ExPolicy &&policy,
  RandomIt first, Sent last, Comp &&comp, 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`.

**Note**  Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons. `comp` has to induce a strict weak ordering on the values.

**Template Parameters**
- **ExPolicy**: The type of the execution policy to 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 `util::projection_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.

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.

Return  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, typename Proj>
hpx::traits::range_iterator<Rng>::type stable_sort(Rng &&rng, Compare &&comp, 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.

Note  Complexity: O(Nlog(N)), where N = std::distance(first, last) comparisons. comp has to induce a strict weak ordering on the values.

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 util::projection_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.

The assignments in the parallel stable_sort algorithm invoked without an execution policy object execute in sequential order in the calling thread.

Return  The stable_sort algorithm returns typename hpx::traits::range_iterator<Rng>::type. It returns last.

```
template<typename ExPolicy, typename Rng, typename Pred, typename Proj>
util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_iterator<Rng>::type>::type stable_sort(ExPolicy &&policy, Rng &&rng, Pred &&pred, Pred &&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
\[
\text{INVOLVE}(\text{comp}, \text{INVOLVE}(\text{proj}, *(i + n)), \text{INVOLVE}(\text{proj}, *i)) == \text{false}.
\]

**Note** Complexity: \( O(N \log(N)) \), where \( N = \text{std::distance}(\text{first}, \text{last}) \) comparisons. \text{comp} has to induce a strict weak ordering on the values.

**Template Parameters**
- **ExPolicy**: The type of the execution 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 \text{util::projection_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**: \text{comp} is a callable object. The return value of the \text{INVOLVE} operation applied to an object of type \text{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 \text{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 \text{comp} is invoked.

The application of function objects in parallel algorithm invoked with an execution policy object of type \text{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 \text{parallel_policy} or \text{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** The \text{stable_sort} algorithm returns a \text{hpx::future<typename \text{hpx::traits::range_iterator<Rng>::type>}} if the execution policy is of type \text{sequenced_task_policy} or \text{parallel_task_policy} and returns \text{typename \text{hpx::traits::range_iterator<Rng>::type}} otherwise. It returns \text{last}.

```cpp
namespace hpx

namespace ranges

Functions

template<
  typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename Pred, typename Proj1, typename Proj2
>
bool starts_with(FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, Pred &&pred, Proj1 &&proj1, Proj2 &&proj2)
Checks whether the second range defined by \([\text{first1}, \text{last1})\) matches the prefix of the first range defined by \([\text{first2}, \text{last2})\)

The assignments in the parallel \text{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 `util::projection_identity`
• **Proj2**: The type of an optional projection function for the destination range. This defaults to `util::projection_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 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.

**Return** 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, typename Proj1, typename Proj2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type
starts_with(ExPolicy &&policy, FwdIter1 first1, Sent1 last1, FwdIter2 first2, Sent2 last2, Pred &&pred, Proj1 &&proj1, 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.

**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 `util::projection_identity`
- **Proj2**: The type of an optional projection function for the destination range. This defaults to `util::projection_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 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.

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.

**Return** The `starts_with` algorithm returns a `hpx::future<bool>` if the execution policy is of type `sequence_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.

```cpp
template<typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
bool starts_with(Rng1 &&rng1, Rng2 &&rng2, Pred &&pred, Proj1 &&proj1, 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 `util::projection_identity`
- **Proj2**: The type of an optional projection function for the destination range. This defaults to `util::projection_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 ele-
ments in the destination range as a projection operation before the actual predicate is invoked.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2, typename Pred, typename Proj1, typename Proj2>
hpx::parallel::util::detail::algorithm_result<ExPolicy, bool>::type starts_with(
    ExPolicy &&policy,
    Rng1   &&rng1,
    Rng2   &&rng2,
    Pred   &&pred,
    Proj1  &&proj1,
    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.

**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 `util::projection_identity`
- `Proj2`: The type of an optional projection function for the destination range. This defaults to `util::projection_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.

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.

**Return** 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.

```cpp
namespace hpx

namespace ranges
```
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.

**Return** 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.

**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.

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.

**Return** 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`.

```cpp
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.

**Return** 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`.

```cpp
template<typename ExPolicy, typename Rng1, typename Rng2>
```
Exchanges elements between range \([\text{first1}, \text{last1})\) and another range starting at \text{first2}.

The swap operations in the parallel \textit{swap_ranges} algorithm invoked with an execution policy object of type \textit{sequenced_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Linear in the distance between \text{first1} and \text{last1}

\textbf{Template Parameters}

- \textbf{ExPolicy}: The type of the execution policy to 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.
- \textbf{Rng1}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \textbf{Rng2}: The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

\textbf{Parameters}

- \textbf{policy}: The execution policy to use for the scheduling of the iterations.
- \textbf{rng1}: Refers to the sequence of elements of the first range.
- \textbf{rng2}: Refers to the sequence of elements of the second range.

The swap operations in the parallel \textit{swap_ranges} algorithm invoked with an execution policy object of type \textit{parallel_policy} or \textit{parallel_task_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return} The \textit{swap_ranges} algorithm returns a \textit{hpx::future<swap_ranges_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>>} if the execution policy is of type \textit{parallel_task_policy} and returns \textit{swap_ranges_result<hpx::traits::range_iterator_t<Rng1>, hpx::traits::range_iterator_t<Rng1>>>}, otherwise. The \textit{swap_ranges} algorithm returns \textit{in_in_result} with the first element as the iterator to the element past the last element exchanged in range beginning with \text{first1} and the second element as the iterator to the element past the last element exchanged in the range beginning with \text{first2}.

\begin{verbatim}
namespace hpx

namespace ranges

\end{verbatim}
Functions

```cpp
template<typename ExPolicy, typename Rng, typename OutIter, typename F, typename Proj = util::projection_identity>
util::detail::algorithm_result<ExPolicy, ranges::unary_transform_result<typename hpx::traits::range_iterator<Rng>::type, OutIter>>::type
transform(
    ExPolicy&& policy,
    Rng&& rng,
    OutIter 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 \( dest \).

The invocations of \( f \) in the parallel \( transform \) algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

**Note**  Complexity: Exactly \( \text{size(rng)} \) applications of \( f \)

**Template Parameters**

- \texttt{ExPolicy}: The type of the execution policy to 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 \( f \).
- \texttt{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- \texttt{OutIter}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an output iterator.
- \texttt{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 \texttt{CopyConstructible}.
- \texttt{Proj}: The type of an optional projection function. This defaults to \texttt{util::projection\_identity}

**Parameters**

- \texttt{policy}: The execution policy to use for the scheduling of the iterations.
- \texttt{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \texttt{dest}: Refers to the beginning of the destination range.
- \texttt{f}: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). 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 \texttt{const\&}. The type \texttt{Type} must be such that an object of type \texttt{range\_iterator<Rng>::type} can be dereferenced and then implicitly converted to \texttt{Type}. The type \texttt{Ret} must be such that an object of type \texttt{OutIter} can be dereferenced and assigned a value of type \texttt{Ret}.

- \texttt{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.

The invocations of \( f \) in the parallel \( transform \) algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.
Return  The `transform` algorithm returns a `hpx::future<ranges::unary_transform_result<range_iterator<Rng>::type, OutIter> >` if the execution policy is of type `parallel_task_policy` and returns `ranges::unary_transform_result<range_iterator<Rng>::type, OutIter>` 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 F, typename Proj = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<ExPolicy, ranges::unary_transform_result<FwdIter1, FwdIter2>> transform(ExPolicy&& policy, 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`.

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.

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 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`.
- **FwdIter2**: The type of the source iterators for the first range 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 `transform` requires `F` to meet the requirements of `CopyConstructible`.
- **Proj**: The type of an optional projection function. This defaults to `util::projection_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.

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename FwdIter1, typename Sent1, typename FwdIter2, typename Sent2, typename FwdIter3, typename F, typename Proj1 = hpx::parallel::util::projection_identity, typename Proj2 = hpx::parallel::util::projection_identity>
parallel::util::detail::algorithm_result<
    ExPolicy,
    ranges::binary_transform_result<FwdIter1, FwdIter2, FwdIter3>
>::type
transform

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.

**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 an 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 an forward iterator.
- **Sent2**: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for FwdIter2.
- **FwdIter3**: The type of the source iterators for the first range 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 `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 `util::projection_identity`.
- **Proj2**: The type of an optional projection function to be used for elements of the second sequence. This defaults to `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
namespace hpx
{
    namespace ranges
    {

638 Chapter 2. What’s so special about HPX?
Functions

template<typename InIter, typename Sent, typename OutIter, typename T, typename BinOp, typename UnOp>
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 \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \( \text{GENERALIZED_NONCOMMUTATIVE_SUM} \) \((\text{binary} \_ \text{op}, \text{init}, \text{conv}(\text{*first}), \ldots, \text{conv}(\text{*first} + (i - \text{result}) - 1))\).

The reduce operations in the parallel transform_exclusive_scan algorithm invoked without an execution policy object execute in sequential order in the calling thread.

**Note** Complexity: \( O(\text{last} - \text{first}) \) applications of the predicates op and conv.

**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.
- **Conv**: 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).
- **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.
- **conv**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a unary predicate. The signature of this predicate should be equivalent to:

\[
\text{R \fun(\text{const Type \&a})};
\]

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.
- **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:

\[
\text{Ret \fun(\text{const Type1 \&a, const Type1 \&b})};
\]

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.
Neither \textit{conv} nor \textit{op} shall invalidate iterators or subranges, or modify elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

**Return** The \textit{transform\textunderscore exclusive\textunderscore scan} algorithm returns \textit{transform\textunderscore exclusive\textunderscore scan\textunderscore result<InIter, OutIter>}. The \textit{transform\textunderscore exclusive\textunderscore 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.

**Note** \textit{GENERALIZED\textunderscore NONCOMMUTATIVE\textunderscore SUM}(\textit{op}, a_1, \ldots, a_N) is defined as:

- a_1 when \(N = 1\)
- \textit{op}(\textit{GENERALIZED\textunderscore NONCOMMUTATIVE\textunderscore SUM}(\textit{op}, a_1, \ldots, a_K), \textit{GENERALIZED\textunderscore NONCOMMUTATIVE\textunderscore SUM}(\textit{op}, a_M, \ldots, a_N)) where \(1 < K+1 = M <= N\).

The behavior of \textit{transform\textunderscore exclusive\textunderscore scan} may be non-deterministic for a non-associative predicate.

```cpp
template<typename ExPolicy, typename FwdIter1, typename Sent, typename FwdIter2, typename T, typename BinOp, typename UnOp>
parallel::util::detail::algorithm_result<ExPolicy, transform_exclusive_result<FwdIter1, FwdIter2>>::type transform_exclusive_scan(
    ExPolicy&& policy,
    FwdIter1 first,
    Sent last,
    FwdIter2 dest,
    T init,
    BinOp&& binary_op,
    UnOp&& unary_op)
```

Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \textit{GENERALIZED\textunderscore NONCOMMUTATIVE\textunderscore SUM}(\textit{binary\textunderscore op}, \textit{init}, \textit{conv}(*\text{first}), \ldots, \textit{conv}(*(\text{first} + (i - \text{result}) - 1))).

The reduce operations in the parallel \textit{transform\textunderscore exclusive\textunderscore scan} algorithm invoked with an execution policy object of type \textit{sequenced\textunderscore policy} execute in sequential order in the calling thread.

**Note** Complexity: \(O(\text{last} - \text{first})\) applications of the predicates \textit{op} and \textit{conv}.

**Template Parameters**

- \textit{ExPolicy}: The type of the execution policy to 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.
- \textit{FwdIter1}: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- \textit{Sent}: The type of the source sentinel (deduced). This sentinel type must be a sentinel for \textit{FwdIter}.
- \textit{FwdIter2}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- \textit{Conv}: The type of the unary function object used for the conversion operation.
- \textit{T}: The type of the value to be used as initial (and intermediate) values (deduced).
- \textit{Op}: The type of the binary function object used for the reduction operation.

**Parameters**

- \textit{policy}: The execution policy to use for the scheduling of the iterations.
- \textit{first}: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- \textit{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.
- **conv**: 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:

  ```cpp
def 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.
- **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:

  ```cpp
def 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.

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 subranges, or modify elements in the ranges [first,last) or [result, result + (last - first)).

**Return** The `transform_exclusive_scan` algorithm returns a `hpx::future<transform_exclusive_result<FwdIter1, FwdIter2>>` if the execution policy is of type `sequenced_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.

**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.

The behavior of `transform-exclusive_scan` may be non-deterministic for a non-associative predicate.

```
template<
    typename Rng,
    typename O,
    typename T,
    typename BinOp,
    typename UnOp>

transform_exclusive_scan_result<
    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.

**Note** Complexity: \( O(\text{last} - \text{first}) \) applications of the predicates \( op \) and \( conv \).

**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).
- **Conv**: 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).
- **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.
- **conv**: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). 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 \( \text{FwdIter1} \) 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.
- **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.

Neither \( conv \) nor \( op \) shall invalidate iterators or subranges, or modify elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first})] \).

**Return** 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.

**Note** `GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aN)` is defined as:

- \( a1 \) when \( N = 1 \)
- \( \text{op}(\text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, a1, \ldots, aK), \text{GENERALIZED_NONCOMMUTATIVE_SUM}(op, aM, \ldots, aN)) \) where \( 1 < K+1 = M \leq N \).

The behavior of `transform_exclusive_scan` may be non-deterministic for a non-associative predicate.
Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \(\text{GENERALIZED_NONCOMMUTATIVE\_SUM}(\text{binary\_op}, \text{init}, \text{conv}(\ast\text{first}), \ldots, \text{conv}(\ast(\text{first} + (i - \text{result}) - 1)))\).

The reduce operations in the parallel \text{transform\_exclusive\_scan} algorithm invoked with an execution policy object of type \text{sequenced\_policy} execute in sequential order in the calling thread.

**Note** Complexity: \(O(\text{last} - \text{first})\) applications of the predicates \text{op} and \text{conv}.

**Template Parameters**
- \text{ExPolicy}: The type of the execution policy to 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.
- \text{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- \text{O}: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.
- \text{Conv}: The type of the unary function object used for the conversion operation.
- \text{T}: The type of the value to be used as initial (and intermediate) values (deduced).
- \text{Op}: The type of the binary function object used for the reduction operation.

**Parameters**
- \text{policy}: The execution policy to use for the scheduling of the iterations.
- \text{rng}: Refers to the sequence of elements the algorithm will be applied to.
- \text{dest}: Refers to the beginning of the destination range.
- \text{conv}: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a unary predicate. The signature of this predicate should be equivalent to:

```cpp
R fun(const Type &a);
```

The signature does not need to have \text{const\&}, but the function must not modify the objects passed to it. The type \text{Type} must be such that an object of type \text{FwdIter1} can be dereferenced and then implicitly converted to \text{Type}. The type \(R\) must be such that an object of this type can be implicitly converted to \(T\).
- \text{init}: The initial value for the generalized sum.
- \text{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:

```cpp
Ret fun(const Type &a, const Type &b);
```

The signature does not need to have \text{const\&}, but the function must not modify the objects
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.

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 subranges, or modify elements in the ranges [first,last) or [result, result + (last - first)).

**Return** 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.

**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.

The behavior of *transform_exclusive_scan* may be non-deterministic for a non-associative predicate.

```cpp
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.

**Note** Complexity: O(last - first) applications of the predicates *op* and *conv*.

**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.
- *Conv*: 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.
• **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:

```cpp
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.

• **conv**: 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:

```cpp
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`.

Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges `[first, last)` or `[result, result + (last - first))`.

**Return** 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.

**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.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

Assigns through each iterator i in `[result, result + (last - first))` the value of `GENERALIZED_NONCOMMUTATIVE_SUM(op, conv(*first), . . . , conv(*(first + (i - result))))`.

```cpp
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_inclusive_scan(ExPolicy &&policy, FwdIter1 first, Sent last, FwdIter2 dest, BinOp &&binary_op, UnOp &&unary_op)
```
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.

**Note** Complexity: \( O(last - first) \) applications of the predicates \( op \) and \( conv \).

**Template Parameters**
- `ExPolicy`: The type of the execution policy to 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.
- `Op`: The type of the binary function object used for the reduction operation.
- `Conv`: 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.
- `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:

  ```cpp
  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.
- `conv`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```cpp
  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`.

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 subranges, or modify elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

**Return** 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.

**Note** GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, \ldots, aN) is defined as:

- \( a1 \) when \( N = 1 \)
\begin{itemize}
  \item \texttt{op(GENERALIZED\_NONCOMMUTATIVE\_SUM(op, a1, \ldots, aK), GENERALIZED\_NONCOMMUTATIVE\_SUM(op, aM, \ldots, aN) where 1 < K+1 = M <= N.}
  
  The behavior of transform\_inclusive\_scan may be non-deterministic for a non-associative predicate.
\end{itemize}

\begin{verbatim}
template<typename Rng, typename O, typename BinOp, typename UnOp>
transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O> transform_inclusive_scan( Rng &&rng, O dest, BinOp &&binary_op, UnOp &&unary_op)
\end{verbatim}

Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of \texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM(op, conv(*first), \ldots, conv(*(first + (i - result))))}.

The reduce operations in the parallel \texttt{transform\_inclusive\_scan} algorithm invoked without an execution policy object execute in sequential order in the calling thread.

\textbf{Note} Complexity: \( O(\text{last} - \text{first}) \) applications of the predicates \texttt{op} and \texttt{conv}.

\textbf{Template Parameters}
\begin{itemize}
  \item \texttt{Rng}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
  \item \texttt{O}: The type of the iterator representing the destination range (deduced).
  \item \texttt{Op}: The type of the binary function object used for the reduction operation.
  \item \texttt{Conv}: The type of the unary function object used for the conversion operation.
\end{itemize}

\textbf{Parameters}
\begin{itemize}
  \item \texttt{rng}: Refers to the sequence of elements the algorithm will be applied to.
  \item \texttt{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:

\begin{verbatim}
Ret fun(const Type1 &a, const Type1 &b);
\end{verbatim}

The signature does not need to have const&., but the function must not modify the objects passed to it. The types \texttt{Type1} and \texttt{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.
  \item \texttt{conv}: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a unary predicate. The signature of this predicate should be equivalent to:

\begin{verbatim}
R fun(const Type &a);
\end{verbatim}

The signature does not need to have const&., but the function must not modify the objects passed to it. The type \texttt{Type} must be such that an object of type \texttt{FwdIter} can be dereferenced and then implicitly converted to \texttt{Type}. The type \texttt{R} must be such that an object of this type can be implicitly converted to \texttt{T}.

Neither \texttt{conv} nor \texttt{op} shall invalidate iterators or subranges, or modify elements in the ranges \([\text{first},\text{last})\) or \([\text{result},\text{result} + (\text{last} - \text{first}))\).

\textbf{Return} The \texttt{transform\_inclusive\_scan} algorithm returns a \texttt{transform\_inclusive\_scan\_result<traits::range\_iterator\_t<Rng>, O>}. The \texttt{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.
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.

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

```cpp
template<typename ExPolicy, typename Rng, typename O, typename BinOp, typename UnOp>
parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type
transform_inclusive_scan(ExPolicy&& policy, 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 with an execution policy object of type sequenced_policy execute in sequential order in the calling thread.

Note Complexity: O(last - first) applications of the predicates op and conv.

**Template Parameters**
- ExPolicy: The type of the execution policy to 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.
- Conv: The type of the unary function object used for the conversion operation.
- 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:

```cpp
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.
- conv: 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:
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.

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 subranges, or modify elements in the ranges [first,last) or [result,result + (last - first)).

**Return** 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.

**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.

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

```c++
R fun(const Type &a);
```

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.

**Note** Complexity: O(last - first) applications of the predicates op and conv.

**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.
- **Conv**: The type of the unary function object used for the conversion operation.
- **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.
• conv: 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.

• 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.

Neither conv nor op shall invalidate iterators or subranges, or modify elements in the ranges [first, last) or [result, result + (last - first)).

**Return** 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.

**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.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

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.

**Note** Complexity: \(O(last - first)\) applications of the predicates \(op\) and \(conv\).

**Template Parameters**
- \(ExPolicy\): The type of the execution policy to 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.
- \(Conv\): The type of the unary function object used for the conversion operation.
- \(Op\): 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.
- \(conv\): 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`.
- \(op\): 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.

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 subranges, or modify elements in the ranges \([first, last)\) or \([result, result + (last - first))\).

**Return** 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.

**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.

The behavior of transform_inclusive_scan may be non-deterministic for a non-associative predicate.

template<typename Rng, typename O, typename BinOp, typename UnOp, typename T> transform_inclusive_scan_result<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 \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), ..., conv(*\((\text{first} + (i - \text{result})))\)).

The reduce operations in the parallel transform_inclusive_scan algorithm invoked without an execution policy object execute in sequential order in the calling thread.

**Note** Complexity: \(O(\text{last} - \text{first})\) applications of the predicates \(op\) and \(conv\).

**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).
- \(Conv\): The type of the unary function object used for the conversion operation.
- \(Op\): 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.
- \(conv\): Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is a unary predicate. The signature of this predicate should be equivalent to:

```plaintext
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 \(T\).
- \(op\): 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:

```plaintext
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.
Neither `conv` nor `op` shall invalidate iterators or subranges, or modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

**Return** 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.

**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`.

The behavior of `transform_inclusive_scan` may be non-deterministic for a non-associative predicate.

```cpp
template<typename ExPolicy, typename Rng, typename O, typename BinOp, typename UnOp, typename T>
parallel::util::detail::algorithm_result<ExPolicy, transform_inclusive_scan_result<traits::range_iterator_t<Rng>, O>>::type
transform_inclusive_scan(std::forward<ExPolicy>(policy), std::forward<Rng>(rng), O dest,
std::forward<BinOp>(binary_op), std::forward<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 with an execution policy object of type `sequenced_policy` execute in sequential order in the calling thread.

**Note** Complexity: `O(last - first)` applications of the predicates `op` and `conv`.

**Template Parameters**
- `ExPolicy`: The type of the execution policy to 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.
- `Conv`: The type of the unary function object used for the conversion operation.
- `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.
- `conv`: 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:
The signature does not need to have const&, but the function must not modify the objects passed to it. The type \textit{Type} must be such that an object of type \textit{FwdIter1} can be dereferenced and then implicitly converted to \textit{Type}. The type \textit{R} must be such that an object of this type can be implicitly converted to \textit{T}.

- \textit{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:

\begin{verbatim}
Ret fun(const Type &a, const Type &b);
\end{verbatim}

The signature does not need to have const&, but the function must not modify the objects passed to it. The types \textit{Type1} and \textit{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.

- \textit{init}: The initial value for the generalized sum.

The reduce operations in the parallel \texttt{transform\_inclusive\_scan} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Neither \textit{conv} nor \textit{op} shall invalidate iterators or subranges, or modify elements in the ranges \([\text{first},\text{last})\) or \([\text{result},\text{result} + (\text{last} - \text{first}))\).

**Return** The \texttt{transform\_inclusive\_scan} algorithm returns a \texttt{hp\!x::future<transform\_inclusive\_scan\_result<traits::range\_iterator\_t<Rng>, O>>} if the execution policy is of type \texttt{sequenced\_task\_policy} or \texttt{parallel\_task\_policy} and returns \texttt{transform\_inclusive\_scan\_result<traits::range\_iterator\_t<Rng>, O>} otherwise. The \texttt{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.

**Note** GENERALIZED\_NONCOMMUTATIVE\_SUM\(\textit{op}, \textit{a}1, \ldots, \textit{aN}\) is defined as:

- \(\textit{a}1\) when \(\textit{N} = 1\)
- \textit{op}(GENERALIZED\_NONCOMMUTATIVE\_SUM\(\textit{op}, \textit{a}1, \ldots, \textit{a}K\)), GENERALIZED\_NONCOMMUTATIVE\_SUM\(\textit{op}, \textit{a}M, \ldots, \textit{aN}\) where \(1 < \textit{K} = 1 = \textit{M} \leq \textit{N}\).

The behavior of \texttt{transform\_inclusive\_scan} may be non-deterministic for a non-associative predicate.

```cpp
namespace hpx
```

**Functions**

```cpp
template<typename ExPolicy, typename Rng, typename T, typename Reduce, typename Convert>
util::detail::algorithm_result<ExPolicy, T>::type transform_reduce(ExPolicy &&policy, Rng &&rng, T init, Reduce &&red_op, Convert &&conv_op)
```

Returns GENERALIZED\_SUM\(\textit{red\_op}, \textit{init}, \textit{conv\_op}(\textit{*first}), \ldots, \textit{conv\_op}(\textit{*first + (last - first) - 1}))\).

The reduce operations in the parallel \texttt{transform\_reduce} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

**Note** Complexity: \(O(\text{last} - \text{first})\) applications of the predicates \textit{red\_op} and \textit{conv\_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 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).

• **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:

```cpp
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:

```cpp
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`.

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.

**Return** 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).

**Note** GENERALIZED_SUM(op, a1, ..., aN) is defined as follows:

• a1 when N is 1
• \text{op}(\text{GENERALIZED\_SUM}(\text{op}, b_1, \ldots, b_K), \text{GENERALIZED\_SUM}(\text{op}, b_M, \ldots, b_N)), where:
  – \(b_1, \ldots, b_N\) may be any permutation of \(a_1, \ldots, a_N\) and
  – \(1 < K+1 = M \leq N\).

\text{template<typename ExPolicy, typename Rng1, typename FwdIter2, typename T>}
\text{util::detail::algorithm_result<ExPolicy, T>::type transform_reduce (ExPolicy &&policy, Rng1 &&rng1, FwdIter2 first2, T init)}

Returns the result of accumulating \textit{init} with the inner products of the pairs formed by the elements of two ranges starting at \textit{first1} and \textit{first2}.

The operations in the parallel \textit{transform\_reduce} algorithm invoked with an execution policy object of type \textit{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note}  Complexity: \(O(last - first)\) applications of the predicate \textit{op2}.

\textbf{Template Parameters}

• \textit{ExPolicy}: The type of the execution policy to 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.

• \textit{Rng1}: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

• \textit{FwdIter2}: The type of the second source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.

• \textit{T}: The type of the value to be used as return) values (deduced).

\textbf{Parameters}

• \textit{policy}: The execution policy to use for the scheduling of the iterations.

• \textit{rng1}: Refers to the sequence of elements the algorithm will be applied to.

• \textit{first2}: Refers to the beginning of the second sequence of elements the result will be calculated with.

• \textit{init}: The initial value for the sum.

The operations in the parallel \textit{transform\_reduce} algorithm invoked with an execution policy object of type \textit{parallel\_policy} or \textit{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

\textbf{Return}  The \textit{transform\_reduce} algorithm returns a \textit{hpx::future<T>} if the execution policy is of type \textit{sequenced\_task\_policy} or \textit{parallel\_task\_policy} and returns \textit{T} otherwise.

\text{template<typename ExPolicy, typename Rng1, typename FwdIter2, typename T, typename Reduce, typename Convert util::detail::algorithm_result<ExPolicy, T>::type transform_reduce (ExPolicy &&policy, Rng1 &&rng1, FwdIter2 first2, T init, Reduce &&red_op, Convert &&conv_op)}

Returns the result of accumulating \textit{init} with the inner products of the pairs formed by the elements of two ranges starting at \textit{first1} and \textit{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.

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.

- Rng1: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

- FwdIter2: 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.

- rng1: 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:

```cpp
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

```cpp
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.

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.

Return 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.
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.
• 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.

Return: 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_copy(ExPolicy&& policy, FwdIter1 first1, Sent1 last1, FwdIter2 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 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**

- `ExPolicy`: The type of the execution policy to 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.

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.

**Return** 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.

```cpp
template<
    typename Rng1,
    typename Rng2>

hpx::parallel::util::in_out_result<
    typename hpx::traits::range_traits<Rng1>::iterator_type,
    typename hpx::traits::range_traits<Rng2>::iterator_type>

uninitialized_copy(
    Rng1&& rng1,
    Rng2&& rng2)
```

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
The `uninitialized_copy` algorithm returns an `in_out_result<iterator_type, iterator_type>` where `iterator_type` is deduced from the ranges. 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>::iterator_type, typename hpx::traits::range_traits<Rng2>::iterator_type>> uninitialized_copy(ExPolicy&& policy, Rng1&& rng1, Rng2&& rng2)
```

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.

**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

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.

```
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 ob-
ject 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 \( \text{first} \) 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.

**Return** 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.

```cpp
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_copy_n(ExPolicy&& policy, FwdIter1 first1, Size count, FwdIter2 first2, Sent2 last2)
```

Copies the elements in the range \([\text{first}, \text{first} + \text{count})\), starting from \(\text{first} \) and proceeding to \(\text{first} + \text{count} - 1\), to another range beginning at \(\text{dest} \). 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**
- **ExPolicy**: The type of the execution policy to 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.
• last1: Refers to sentinel value denoting the end of the second range the algorithm will be
  applied to.

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.

Return 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.

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 stor-

age 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 execu-
tion 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 an 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.

Return 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type uninitialized_default_construct
(ExPolicy &&policy,
 FwdIter first,
 Sent last)

Constructs objects of type typename iterator_traits<ForwardIt> ::value_type in the uninitialized stor-
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.

**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 an 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.

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.

**Return** 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.

```cpp
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.

**Return** 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.

```cpp
template<typename ExPolicy, typename 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 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**
- `ExPolicy`: The type of the execution policy to 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

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.

**Return** 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 an 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.

**Return** 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.
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.

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 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.

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.

Return 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.

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 an 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.

**Return** 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.

```cpp
template<
    typename ExPolicy, 
    typename FwdIter, 
    typename Sent
>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type
    uninitialized_fill (ExPolicy
    &&pol-
    icy,
    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 parallel `uninitialized_fill` algorithm invoked with an execution policy object of type `sequenced_policy` execute in sequential order in the calling 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 an 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.

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.

**Return** 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.

```cpp
template<
    typename Rng, 
    typename T
>
```
```cpp
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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng, typename T>
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type uninitialized_fill (ExPolicy &&policy, 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 with an execution policy object of type `sequenced_policy` execute in sequential order in the calling 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.

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.

**Return** The `uninitialized_fill` algorithm returns a `hpx::future<typename hpx::traits::range_traits<Rng>::iterator_type>`, if the execution policy is of type...
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.

Return The uninitialized_fill_n algorithm returns a returns FwdIter. The uninitialized_fill_n algo-
    rithm 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>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type
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.

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.

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.

**Return** 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.

```cpp
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 `InIter`.

**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 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.

**Return** 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_move(
    ExPolicy&& policy, FwdIter1 first1, Sent1 last1, FwdIter2 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 \texttt{uninitialized\_move} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling 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.

The assignments in the parallel \texttt{uninitialized\_move} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

**Return** The \texttt{uninitialized\_move} algorithm returns a \texttt{hpx::future\<in\_out\_result\<InIter, FwdIter\>\>}, if the execution policy is of type \texttt{sequenced\_task\_policy} or \texttt{parallel\_task\_policy} and returns \texttt{in\_out\_result\<InIter, FwdIter\>} otherwise. The \texttt{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.
Moves the elements in the range, defined by \([\text{first}, \text{last})\), to an uninitialized memory area beginning at \(\text{dest}\). If an exception is thrown during the initialization, some objects in \([\text{first}, \text{last})\) are left in a valid but unspecified state.

The assignments in the parallel \texttt{uninitialized\_move} algorithm invoked without an execution policy object will execute in sequential order in the calling thread.

\textbf{Note} Complexity: Performs exactly \texttt{last - first} assignments.

\textbf{Template Parameters}

- \texttt{Rng1} \(\text{The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.}\)
- \texttt{Rng2} \(\text{The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.}\)

\textbf{Parameters}

- \texttt{rng1} \(\text{Refers to the range from which the elements will be moved from}\)
- \texttt{rng2} \(\text{Refers to the range to which the elements will be moved to}\)

\textbf{Return} The \texttt{uninitialized\_move} algorithm returns an \texttt{in\_out\_result\langle\text{typename hpx::traits::range_traits<Rng1>::iterator\_type, typename hpx::traits::range_traits<Rng2>::iterator\_type\rangle}}. The \texttt{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.

\begin{verbatim}
template<\text{typename ExPolicy, typename Rng1, typename Rng2}>
parallel::util::detail::algorithm_result<ExPolicy, hpx::parallel::util::in_out_result<\text{typename hpx::traits::range_traits<Rng1>::iterator\_type, \text{typename hpx::traits::range_traits<Rng2>::iterator\_type\rangle\rangle\rangle\rangle\rangle\rangle
\end{verbatim}

Moves the elements in the range, defined by \([\text{first}, \text{last})\), to an uninitialized memory area beginning at \(\text{dest}\). If an exception is thrown during the initialization, some objects in \([\text{first}, \text{last})\) are left in a valid but unspecified state.

The assignments in the parallel \texttt{uninitialized\_move} algorithm invoked with an execution policy object of type \texttt{sequenced\_policy} execute in sequential order in the calling thread.

\textbf{Note} Complexity: Performs exactly \texttt{last - first} assignments.

\textbf{Template Parameters}

- \texttt{ExPolicy} \(\text{The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.}\)
- \texttt{Rng1} \(\text{The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.}\)
- \texttt{Rng2} \(\text{The type of the destination range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.}\)

\textbf{Parameters}

- \texttt{policy} \(\text{The execution policy to use for the scheduling of the iterations.}\)
- \texttt{rng1} \(\text{Refers to the range from which the elements will be moved from}\)
- \texttt{rng2} \(\text{Refers to the range to which the elements will be moved to}\)

The assignments in the parallel \texttt{uninitialized\_move} algorithm invoked with an execution policy object of type \texttt{parallel\_policy} or \texttt{parallel\_task\_policy} are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.
Return 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.

Return 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>> 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.

**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 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.
- `last1`: Refers to sentinel value denoting the end of the second range the algorithm will be applied to.

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.

**Return** 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.

```cpp
namespace hpx
{
namespace ranges
{

Functions

```

```cpp

```

```cpp
```

```cpp
```

```cpp
```

```cpp
```

```cpp
```

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 an forward iterator.
- `Sent`: The type of the source sentinel (deduced). This sentinel type must be a sentinel for `FwdIter`.  

```cpp
```
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.

Return

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
>
parallel::util::detail::algorithm_result<
  ExPolicy
>::type 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.

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.

**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 an 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.

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.

Return

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.

**Return** 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.

```cpp
template<typename ExPolicy, typename Rng>
parallel::util::detail::algorithm_result<ExPolicy, typename hpx::traits::range_traits<Rng>::iterator_type>::type
uninitialized_value_construct(ExPolicy&& policy, 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 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**
- **ExPolicy**: The type of the execution policy to 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 constructed.

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.

**Return** 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.

```cpp
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 \( \text{count} \) assignments, if \( \text{count} > 0 \), 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 \( f \) 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 \( \text{first} \) the algorithm will be applied to.

**Return** The `uninitialized_value_construct_n` algorithm returns a returns \( \text{FwdIter} \). The `uninitialized_value_construct_n` algorithm returns the iterator to the element in the source range, one past the last element constructed.

```cpp
template<typename ExPolicy, typename FwdIter, typename Size>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter>::type 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 \( [\text{first}, \text{first} + \text{count}) \) 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.  

**Note** Complexity: Performs exactly \( \text{count} \) assignments, if \( \text{count} > 0 \), 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 \( \text{first} \) the algorithm will be applied to.

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.

**Return** 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 \( \text{FwdIter} \) otherwise. The `uninitialized_value_construct_n` algorithm returns the iterator to the element in the source range, one past the last element constructed.

```cpp
namespace hpx
{

namespace ranges
{
```

676 Chapter 2. What’s so special about HPX?
Functions

template<typename FwdIter, typename Sent, typename Pred, typename Proj>
subrange_t<FwdIter, Sent> unique (FwdIter first, Sent last, Pred &&pred, 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 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 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 util::projection_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.
• 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.

Return 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, typename Proj>
parallel::util::detail::algorithm_result<ExPolicy, subrange_t<FwdIter, Sent>>::type unique (ExPolicy &&policy, FwdIter first, Sent last, Pred &&pred, 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.

**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 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 `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 `util::projection_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:

```cpp
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.

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.

**Return** 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.

```cpp
template<typename Rng, typename Pred, typename Proj>
subrange_t<hpx::traits::range_iterator_t<Rng>, hpx::traits::range_iterator_t<Rng>> unique(Rng &&rng, Pred &&pred, 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 \( \text{pred} \) and no more than twice as many applications of the projection \( \text{proj} \), where \( N = \text{std::distance} (\text{begin(rng)}, \text{end(rng)}) \).

Template Parameters
- \( \text{Rng} \): The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- \( \text{Pred} \): The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of \text{unique} requires \( \text{Pred} \) to meet the requirements of \text{CopyConstructible}. This defaults to \text{std::equal_to<>}
- \( \text{Proj} \): The type of an optional projection function. This defaults to \text{util::projection_identity}

Parameters
- \( \text{rng} \): Refers to the sequence of elements the algorithm will be applied to.
- \( \text{pred} \): Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by \([\text{first}, \text{last})\). This is an binary predicate which returns \text{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 \text{const&}, but the function must not modify the objects passed to it. The type \text{Type} must be such that an object of type \text{FwdIter1} can be dereferenced and then implicitly converted to \text{Type}.
- \( \text{proj} \): Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate \text{is} invoked.

Return  The \text{unique} algorithm returns \text{subrange_t<typename hpx::traits::range_iterator<Rng>::type,hpx::traits::range_iterator_t<Rng>>}. The \text{unique} algorithm returns an object \{\text{ret}, \text{last}\}, where \text{ret} is a past-the-end iterator for a new subrange.

```

namespace hpx

namespace parallel

namespace util

template<typename T = detail::no_data, typename Pred = std::less_equal<T>>
class cancellation_token

Public Functions

- \text{cancellation_token} (T \text{data})
- bool \text{was_cancelled} (T \text{data}) const
- void \text{cancel} (T \text{data})
- T \text{get_data} () const

```
Private Types

typedef std::atomic<T> flag_type

Private Members

std::shared_ptr<flag_type> was_cancelled_

template<typename Compare>
struct compare_projected<Compare, util::projection_identity>

Public Functions

template<typename Compare_,
        Compare_&& comp, util::projection_identity>
constexpr compare_projected (Compare_ &&comp, util::projection_identity)

template<typename T1, typename T2>
constexpr bool operator() (T1 &&t1, T2 &&t2) const

Public Members

Compare comp_

template<typename Compare, typename Proj2>
struct compare_projected<Compare, util::projection_identity, Proj2>

Public Functions

template<typename Compare_, typename Proj2_,
        Compare_&& comp, util::projection_identity, Proj2_&&proj2>
constexpr compare_projected (Compare_ &&comp, util::projection_identity, Proj2_ &&proj2)

template<typename T1, typename T2>
constexpr bool operator() (T1 &&t1, T2 &&t2) const

Public Members

Compare comp_

Proj2 proj2_

template<typename Compare, typename Proj1>
struct compare_projected<Compare, Proj1, util::projection_identity>
Public Functions

template<typename Compare_, typename Proj1_>
constexpr compare_projected(Compare_ &comp, Proj1_ &proj1, util::projection_identity) const

template<typename T1, typename T2>
constexpr bool operator()(T1 &t1, T2 &t2) const

Public Members

Compare comp_
Proj1 proj1_

namespace hpx

namespace parallel

namespace util

template<typename Compare, typename Proj>
struct compare_projected<Compare, Proj>

Public Functions

template<typename Compare_, typename Proj_>
constexpr compare_projected(Compare_ &comp, Proj_ &proj)

template<typename T1, typename T2>
constexpr bool operator()(T1 &t1, T2 &t2) const
**Public Members**

Compare `comp_`  
Proj `proj_`

template<typename `Compare`, typename `Proj1`, typename `Proj2`>
struct `compare_projected`<`Compare`, `Proj1`, `Proj2`>

**Public Functions**

template<typename `Compare_`, typename `Proj1_`, typename `Proj2_`>
constexpr `compare_projected`(`Compare_`&& `comp`, `Proj1_`&& `proj1`, `Proj2_`&& `proj2`) const

template<typename `T1`, typename `T2`>
constexpr bool operator()(`T1`&& `t1`, `T2`&& `t2`) const

**Public Members**

Compare `comp_`  
Proj1 `proj1_`  
Proj2 `proj2_`

template<typename `Compare`, typename `Proj1`>
struct `compare_projected`<`Compare`, `Proj1`, `util::projection_identity`>

**Public Functions**

template<typename `Compare_`, typename `Proj1_`>
constexpr `compare_projected`(`Compare_`&& `comp`, `Proj1_`&& `proj1`, `util::projection_identity`) const

template<typename `T1`, typename `T2`>
constexpr bool operator()(`T1`&& `t1`, `T2`&& `t2`) const

**Public Members**

Compare `comp_`  
Proj1 `proj1_`

template<typename `Compare`>
struct `compare_projected`<`Compare`, `util::projection_identity`>
Public Functions

template<typename Compare_>
constexpr compare_projected(Compare_&& comp, util::projection_identity) const

template<typename T1, typename T2>
constexpr bool operator()(T1 &t1, T2 &t2) const

Public Members

Compare comp_

namespace hpx

namespace parallel

2.8. API reference
namespace util

template<
typename Pred,
typename Proj>
struct invoke_projected

Public Types

template<>
using pred_type = typename std::decay<Pred>::type
template<>
using proj_type = typename std::decay<Proj>::type

Public Functions

template<
typename Pred__, typename Proj__>
invoke_projected(Pred__&& pred, Proj__&& proj)
template<
typename T>
dcltype(auto) operator() (T &&t)
template<
typename T>
dcltype(auto) operator() (T &&t, T &&u)

Public Members

pred_type pred_
proj_type proj_

template<
typename Pred>
struct invoke_projected<Pred, projection_identity>

Public Types

template<>
using pred_type = typename std::decay<Pred>::type

Public Functions

template<
typename Pred__>
invoke_projected(Pred__&& pred, projection_identity)
template<
typename T>
dcltype(auto) operator() (T &&t)
template<
typename T>
bool operator() (T &&t, T &&u)
Public Members
pred_type pred_

namespace hpx
namespace parallel
namespace util

Functions

template<typename Iter, typename F, typename Cleanup>
constexpr Iter loop_with_cleanup (Iter it, Iter last, F &&f, Cleanup &&cleanup)

template<typename Iter, typename FwdIter, typename F, typename Cleanup>
constexpr FwdIter loop_with_cleanup (Iter it, Iter last, FwdIter dest, F &&f, Cleanup &&cleanup)

template<typename Iter, typename F, typename Cleanup>
constexpr Iter loop_with_cleanup_n (Iter it, std::size_t count, F &&f, Cleanup &&cleanup)

template<typename Iter, typename FwdIter, typename F, typename Cleanup>
constexpr FwdIter loop_with_cleanup_n (Iter it, std::size_t count, FwdIter dest, F &&f, Cleanup &&cleanup)

template<typename Iter, typename CancelToken, typename F, typename Cleanup>
constexpr Iter loop_with_cleanup_n_with_token (Iter it, std::size_t count, CancelToken &&tok, F &&f, Cleanup &&cleanup)

template<typename Iter, typename FwdIter, typename CancelToken, typename F, typename Cleanup>
constexpr FwdIter loop_with_cleanup_n_with_token (Iter it, std::size_t count, FwdIter dest, CancelToken &&tok, F &&f, Cleanup &&cleanup)

template<typename Iter, typename F>
constexpr Iter loop_idx_n (std::size_t base_idx, Iter it, std::size_t count, F &&f)

template<typename Iter, typename CancelToken, typename F>
constexpr Iter loop_idx_n (std::size_t base_idx, Iter it, std::size_t count, CancelToken &&tok, F &&f)

T accumulate_n (Iter it, std::size_t count, T init, Pred &&f)

T accumulate (Iter first, Iter last, Reduce &&r, Conv &&conv = util::projection_identity)

T accumulate (Iter1 first1, Iter1 last1, Iter2 first2, Reduce &&r, Conv &&conv)
Variables

```cpp
template<typename ExPolicy> HPX_INLINE_CONSTEXPR_VARIABLE loop_step_t<ExPolicy> hpx::parallel::util::loop_step = loop_step_t{};
```

```cpp
template<typename ExPolicy> HPX_INLINE_CONSTEXPR_VARIABLE loop_optimization_t<ExPolicy> hpx::parallel::util::loop_optimization = loop_optimization_t{};
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE loop_t hpx::parallel::util::loop = loop_t{};
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE loop_ind_t hpx::parallel::util::loop_ind = loop_ind_t{};
```

```cpp
template<typename ExPolicy> HPX_INLINE_CONSTEXPR_VARIABLE loop2_t<ExPolicy> hpx::parallel::util::loop2 = loop2_t<ExPolicy>{};
```

```cpp
template<typename ExPolicy> HPX_INLINE_CONSTEXPR_VARIABLE loop_n_t<ExPolicy> hpx::parallel::util::loop_n = loop_n_t<ExPolicy>{};
```

```cpp
template<typename ExPolicy> HPX_INLINE CONSTEXPR_VARIABLE loop_n_ind_t<ExPolicy> hpx::parallel::util::loop_n_ind = loop_n_ind_t<ExPolicy>{};
```

```cpp
template<typename ExPolicy>
struct loop2_t : public hpx::functional::detail::tag_fallback<loop2_t<ExPolicy>>
```

Friends

```cpp
template<typename VecOnly, typename Begin1, typename End1, typename Begin2, typename F>
friend constexpr std::pair<Begin1, Begin2> tag_fallback_invoke(hpx::parallel::util::loop2_t<ExPolicy>,
    VecOnly&&, Begin1 begin1, End1 end1, Begin2 begin2, F &&f)
```

```cpp
struct loop_ind_t : public hpx::functional::detail::tag_fallback<loop_ind_t>
```

Friends

```cpp
template<typename ExPolicy, typename Begin, typename End, typename F>
friend constexpr Begin tag_fallback_invoke(hpx::parallel::util::loop_ind_t,
    ExPolicy&&, Begin begin, End end, F &&f)
```

```cpp
template<typename ExPolicy, typename Begin, typename End, typename CancelToken, typename F>
friend constexpr Begin tag_fallback_invoke(hpx::parallel::util::loop_ind_t,
    ExPolicy&&, Begin begin, End end, CancelToken &tok, F &&f)
```

```cpp
template<typename ExPolicy>
struct loop_n_ind_t : public hpx::functional::detail::tag_fallback<loop_n_ind_t<ExPolicy>>
```
Friends

template<typename Iter, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_ind_t<ExPolicy>,
Iter it, std::size_t count, F &&f)

template<typename Iter, typename CancelToken, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_ind_t<ExPolicy>,
Iter it, std::size_t count, CancelToken &tok, F &&f)

template<typename ExPolicy>
struct loop_n_t : public hpx::functional::detail::tag_fallback<loop_n_t<ExPolicy>>

Friends

template<typename Iter, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_t<ExPolicy>,
Iter it, std::size_t count, F &&f)

template<typename Iter, typename CancelToken, typename F>
friend constexpr Iter tag_fallback_invoke (hpx::parallel::util::loop_n_t<ExPolicy>,
Iter it, std::size_t count, CancelToken &tok, F &&f)

template<typename ExPolicy>
struct loop_optimization_t : public hpx::functional::detail::tag_fallback<loop_optimization_t<ExPolicy>>

Friends

template<typename Iter>
friend constexpr bool tag_fallback_invoke (hpx::parallel::util::loop_optimization_t<ExPolicy>,
Iter, Iter)

template<typename ExPolicy>
struct loop_step_t : public hpx::functional::detail::tag_fallback<loop_step_t<ExPolicy>>

Friends

template<typename VecOnly, typename F, typename ...Iters>
hpx::util::invoke_result<F, Iters...>::type tag_fallback_invoke (hpx::parallel::util::loop_step_t<ExPolicy>,
VecOnly&&, F &&f, Iters&&... its)

struct loop_t : public hpx::functional::detail::tag_fallback<loop_t>
Friends

template<typename ExPolicy, typename Begin, typename End, typename F>
friend constexpr Begin tag_fallback_invoke(hpx::parallel::util::loop_t, ExPolicy&&, Begin begin, End end, F &&f)

template<typename ExPolicy, typename Begin, typename End, typename CancelToken, typename F>
friend constexpr Begin tag_fallback_invoke(hpx::parallel::util::loop_t, ExPolicy&&, Begin begin, End end, CancelToken &tok, F &&f)

namespace hpx

namespace parallel

namespace util

Functions

template<typename Value, typename ...Args>
void construct_object (Value *ptr, Args&&... args)
create an object in the memory specified by ptr

Template Parameters
• Value:: typename of the object to create
• Args:: parameters for the constructor

Parameters
• [in] ptr:: pointer to the memory where to create the object
• [in] args:: arguments to the constructor

template<typename Value>
void destroy_object (Value *ptr)
destroy an object in the memory specified by ptr

Template Parameters
• Value:: typename of the object to create

Parameters
• [in] ptr:: pointer to the object to destroy

template<typename Iter, typename Sent>
void init (Iter first, Sent last, typename std::iterator_traits<Iter>::value_type &val)
Initialize a range of objects with the object val moving across them

Return range initialized

Parameters
• [in] r:: range of elements not initialized
• [in] val:: object used for the initialization

template<typename Value, typename ...Args>
void construct (Value *ptr, Args&&... args)
create an object in the memory specified by ptr
Template Parameters
- Value: typename of the object to create
- Args: parameters for the constructor

Parameters
- [in] ptr: pointer to the memory where to create the object
- [in] args: arguments to the constructor

```cpp
template<typename Iter1, typename Sent1, typename Iter2>
Iter2 init_move(Iter2 it_dest, Iter1 first, Sent1 last)
Move objects.
```

Template Parameters
- Iter: iterator to the elements
- Value: typename of the object to create

Parameters
- [in] itdest: iterator to the final place of the objects
- [in] R: range to move

```cpp
template<typename Iter, typename Sent, typename Value = typename std::iterator_traits<Iter>::value_type>
Value *uninit_move(Value *ptr, Iter first, Sent last)
Move objects to uninitialized memory.
```

Template Parameters
- Iter: iterator to the elements
- Value: typename of the object to construct

Parameters
- [in] ptr: pointer to the memory where to create the object
- [in] R: range to move

```cpp
template<typename Iter, typename Sent>
void destroy(Iter first, Sent last)
Move objects to uninitialized memory.
```

Template Parameters
- Iter: iterator to the elements
- Value: typename of the object to construct

Parameters
- [in] ptr: pointer to the memory where to construct the object
- [in] R: range to move

```cpp
template<typename Iter1, typename Sent1, typename Iter2, typename Compare>
Iter2 full_merge(Iter1 buf1, Sent1 end_buf1, Iter1 buf2, Sent1 end_buf2, Iter2 buf_out, Compare comp)
Merge two contiguous buffers pointed by buf1 and buf2, and put in the buffer pointed by buf_out.
```

Parameters
- [in] buf1: iterator to the first element in the first buffer
- [in] end_buf1: final iterator of first buffer
- [in] buf2: iterator to the first iterator to the second buffer
- [in] end_buf2: final iterator of the second buffer
- [in] buf_out: buffer where move the elements merged
- [in] comp: comparison object

```cpp
template<typename Iter, typename Sent, typename Value, typename Compare>
```
Value *uninit_full_merge (Iter first1, Sent last1, Iter first2, Sent last2, Value *it_out, Compare comp)

Merge two contiguous buffers pointed by first1 and first2, and put in the uninitialized buffer pointed by it_out.

Parameters
• [in] first1:: iterator to the first element in the first buffer
• [in] last:: last iterator of the first buffer
• [in] first2:: iterator to the first element to the second buffer
• [in] last2:: final iterator of the second buffer
• [in] it_out:: uninitialized buffer where move the elements merged
• [in] comp:: comparison object

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
Iter2 half_merge (Iter1 buf1, Sent1 end_buf1, Iter2 buf2, Sent2 end_buf2, Iter2 buf_out, Compare comp)

: Merge two buffers. The first buffer is in a separate memory. The second buffer have a empty space before buf2 of the same size than the (end_buf1 - buf1)

Remark  The elements pointed by Iter1 and Iter2 must be the same
Parameters
• [in] buf1:: iterator to the first element of the first buffer
• [in] end_buf1:: iterator to the last element of the first buffer
• [in] buf2:: iterator to the first element of the second buffer
• [in] end_buf2:: iterator to the last element of the second buffer
• [in] buf_out:: iterator to the first element to the buffer where put the result
• [in] comp:: object for Compare two elements of the type pointed by the Iter1 and Iter2

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Compare>
bool in_place_merge_uncontiguous (Iter1 src1, Sent1 end_src1, Iter2 src2, Sent2 end_src2, Iter3 aux, Compare comp)

Merge two non contiguous buffers, placing the results in the buffers for to do this use an auxiliary buffer pointed by aux
Parameters
• [in] src1:: iterator to the first element of the first buffer
• [in] end_src1:: last iterator of the first buffer
• [in] src2:: iterator to the first element of the second buffer
• [in] end_src2:: last iterator of the second buffer
• [in] aux:: iterator to the first element of the auxiliary buffer
• [in] comp:: object for to Compare elements

Exceptions
•

template<typename Iter1, typename Sent1, typename Iter2, typename Compare>
bool in_place_merge (Iter1 src1, Iter1 src2, Sent1 end_src2, Iter2 buf, Compare comp)

: merge two contiguous buffers, using an auxiliary buffer pointed by buf
Parameters
• [in] src1:: iterator to the first position of the first buffer
• [in] src2:: final iterator of the first buffer and first iterator of the second buffer
• [in] end_src2:: final iterator of the second buffer
• [in] buf:: iterator to buffer used as auxiliary memory
• [in] comp:: object for to Compare elements

Exceptions
namespace hpx

namespace parallel

namespace util

Functions

template<typename Iter, typename Sent, typename Compare>
bool less_range(Iter it1, std::uint32_t pos1, Sent it2, std::uint32_t pos2, Compare comp)
    Compare the elements pointed by it1 and it2, and if they are equals, compare their position, doing a stable comparison.

Return result of the comparison

Parameters
• [in] it1: iterator to the first element
• [in] pos1: position of the object pointed by it1
• [in] it2: iterator to the second element
• [in] pos2: position of the element pointed by it2
• [in] comp: comparison object

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
util::range<Iter1, Sent1> full_merge4(utility::range<Iter1, Sent1> &dest, utility::range<Iter2, Sent2> vrange_input[4], std::uint32_t nrange_input, Compare comp)

Merge four ranges.

Return range with all the elements move with the size adjusted

Parameters
• [in] dest: range where move the elements merged. Their size must be greater or equal than the sum of the sizes of the ranges in the array R
• [in] R: array of ranges to merge
• [in] nrange_input: number of ranges in R
• [in] comp: comparison object

template<typename Value, typename Iter, typename Sent, typename Compare>
utility::range<Value*> uninit_full_merge4(utility::range<Value*> const &dest, utility::range<Iter, Sent> vrange_input[4], std::uint32_t nrange_input, Compare comp)

Merge four ranges and put the result in uninitialized memory.

Return range with all the elements move with the size adjusted

Parameters
• [in] dest: range where create and move the elements merged. Their size must be greater or equal than the sum of the sizes of the ranges in the array R
• [in] R: array of ranges to merge
• [in] nrange_input: number of ranges in vrange_input
• [in] comp: comparison object
namespace hpx

namespace parallel

namespace util

Functions

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
void merge_level4 (util::range<Iter1, Sent1> dest, std::vector<util::range<Iter2, Sent2>>& v_input, std::vector<util::range<Iter1, Sent1>>& v_output, Compare comp)
Merge the ranges in the vector v_input using full_merge4. The v_output vector is used as auxiliary memory in the internal process. The final results is in the dest range. All the ranges of v_output are inside the range dest
Return range with all the elements moved
Parameters
• [in] dest:: range where move the elements merged
• [in] v_input:: vector of ranges to merge
• [in] v_output:: vector of ranges obtained
• [in] comp:: comparison object

template<typename Value, typename Iter, typename Sent, typename Compare>
void uninit_merge_level4 (util::range<Value*> dest, std::vector<util::range<Iter, Sent>>& v_input, std::vector<util::range<Value*>>& v_output, Compare comp)
Merge the ranges over uninitialized memory in the vector v_input using full_merge4. The v_output vector is used as auxiliary memory in the internal process. The final results is in the dest range. All the ranges of v_output are inside the range dest
Return range with all the elements moved
Parameters
• [in] dest:: range where move the elements merged
• [in] v_input:: vector of ranges to merge
• [in] v_output:: vector of ranges obtained
• [in] comp:: comparison object

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Compare>
util::range<Iter2, Sent2> merge_vector4 (util::range<Iter1, Sent1> range_input, util::range<Iter2, Sent2> range_output, std::vector<util::range<Iter1, Sent1>> &v_input, std::vector<util::range<Iter2, Sent2>> &v_output, Compare comp)
Merge the ranges in the vector v_input using merge_level4. The v_output vector is used as auxiliary memory in the internal process. The final results is in the range_output range. All the ranges of v_output are inside the range range_output All the ranges of v_input are inside the range range_input
Parameters
• [in] range_input:: range including all the ranges of v_input

namespace hpx
namespace parallel

namespace util

Functions

constexpr std::uint32_t nbits32 (std::uint32_t num)
Obtain the number of bits equal or greater than num.

Return Number of bits
Parameters
  • [in] num : Number to examine
Exceptions
  • none:

constexpr std::uint64_t nbits64 (std::uint64_t num)
Obtain the number of bits equal or greater than num.

Return Number of bits
Parameters
  • [in] num : Number to examine
Exceptions
  • none:

Variables

HPX_INLINE_CONSTEXPR_VARIABLE const std::uint32_t hpx::parallel::util::tmsb[256]

namespace hpx

namespace parallel

namespace util

Functions

template<typename Itr, typename ...Ts>
prefetching::prefetcher_context<Itr, Ts const...> make_prefetcher_context (Itr base_begin,
Itr base_end,
std::size_t p_factor,
Ts const&... rngs)

namespace prefetching
Functions

template<typename ...Ts, std::size_t... Is>
void prefetch_containers (hpx::tuple<Ts...> const &t, hpx::util::index_pack<Is...>, std::size_t idx)

template<typename ExPolicy, typename Itr, typename ...Ts, typename F>
constexpr prefetching_iterator<Itr, Ts...> tag_invoke (hpx::parallel::util::loop_n_t<ExPolicy>,
prefetching_iterator<Itr, Ts...> it, std::size_t count, F &&f)

template<typename ExPolicy, typename Itr, typename ...Ts, typename F>
constexpr prefetching_iterator<Itr, Ts...> tag_invoke (hpx::parallel::util::loop_n_ind_t<ExPolicy>,
prefetching_iterator<Itr, Ts...> it, std::size_t count, F &&f)

struct loop_n_helper

Public Static Functions

template<typename Itr, typename ...Ts, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr, Ts...> it, std::size_t count, F &&f, Pred)

template<typename Itr, typename ...Ts, typename CancelToken, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr, Ts...> it, std::size_t count, CancelToken &tok, F &&f, Pred)

struct loop_n_ind_helper

Public Static Functions

template<typename Itr, typename ...Ts, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr, Ts...> it, std::size_t count, F &&f, Pred)

template<typename Itr, typename ...Ts, typename CancelToken, typename F, typename Pred>
static constexpr prefetching_iterator<Itr, Ts...> call (prefetching_iterator<Itr, Ts...> it, std::size_t count, CancelToken &tok, F &&f, Pred)

template<typename Itr, typename ...Ts>
struct prefetcher_context
Public Functions

`prefetcher_context (Itr begin, Itr end, ranges_type const &rngs, std::size_t p_factor = 1)`

`prefetching_iterator<Itr, Ts...> begin ()`

`prefetching_iterator<Itr, Ts...> end ()`

Private Types

`typedef hpx::tuple<std::reference_wrapper<Ts>...> ranges_type`

Private Members

`Itr it_begin_`

`Itr it_end_`

`ranges_type rngs_`

`std::size_t chunk_size_`

`std::size_t range_size_`

Private Static Attributes

`constexpr std::size_t sizeof_first_value_type = sizeof(typename hpx::tuple_element<0, ranges_type>::type::type)`

```
template<typename Itr, typename ...Ts> class prefetching_iterator
```

Public Types

```
template<> using base_iterator = Itr
```

```
template<> using iterator_category = std::random_access_iterator_tag
```

```
template<> using value_type = typename std::iterator_traits::value_type
```

```
template<> using difference_type = std::ptrdiff_t
```

```
template<> using pointer = value_type*
```

```
template<> using reference = value_type&
```
Public Functions

`prefetching_iterator` *(std::size_t idx, base_iterator base, std::size_t chunk_size, std::size_t range_size, ranges_type const &rngs)*

ranges_type const &ranges() const

Itr base() const

std::size_t chunk_size() const

std::size_t range_size() const

std::size_t index() const

prefetching_iterator &operator+=(difference_type rhs)

prefetching_iterator &operator-=(difference_type rhs)

prefetching_iterator &operator++()

prefetching_iterator &operator--()

prefetching_iterator operator++(int)

prefetching_iterator operator--(int)

difference_type operator-(const prefetching_iterator &rhs) const

bool operator==(const prefetching_iterator &rhs) const

bool operator!=(const prefetching_iterator &rhs) const

bool operator>(const prefetching_iterator &rhs) const

bool operator<(const prefetching_iterator &rhs) const

bool operator>=(const prefetching_iterator &rhs) const

bool operator<=(const prefetching_iterator &rhs) const

`std::size_t &operator[](std::size_t)`

`std::size_t operator*() const`

Private Types

template<>

**using ranges_type = hpx::tuple<std::reference_wrapper<Ts>...>**
**Private Members**

ranges_type `rngs_
base_iterator `base_
std::size_t `chunk_size_
std::size_t `range_size_
std::size_t `idx_

**Friends**
prefetching_iterator `operator+ (prefetching_iterator const &lhs, difference_type rhs)
prefetching_iterator `operator- (prefetching_iterator const &lhs, difference_type rhs)

namespace hpx

**Typedefs**

using `identity = hpx::parallel::util::projection_identity

namespace parallel

namespace util

struct `projection_identity

**Public Types**

using `is_transparent = std::true_type

**Public Functions**

template<typename T>
`constexpr T &&operator() (T &&val) const

namespace hpx

namespace parallel

namespace util
**Typedefs**

```cpp
template<typename Iterator, typename Sentinel = Iterator>
using range = hpx::util::iterator_range<Iterator, Sentinel>
```

**Functions**

```cpp
template<typename Iter, typename Sent>
range<Iter, Sent> concat (range<Iter, Sent> const &it1, range<Iter, Sent> const &it2)
```

> **Return** range resulting of the concatenation

**Parameters**
- **[in]** `it1`: first range
- **[in]** `it2`: second range

```cpp
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`.

**Return** range with the objects moved and the size adjusted

**Parameters**
- **[in]** `dest`: range where move the objects
- **[in]** `src`: range from where move the objects

```cpp
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`.

**Return** range with the objects moved and the size adjusted

**Parameters**
- **[in]** `dest`: range where move and create the objects
- **[in]** `src`: range from where move the objects

```cpp
template<typename Iter, typename Sent>
void destroy_range (range<Iter, Sent> r)
```

> destroy a range of objects

**Parameters**
- **[in]** `r`: range to destroy

```cpp
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

**Return** range initialized

**Parameters**
- **[in]** `r`: range of elements not initialized
- **[in]** `val`: object used for the initialization
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

Parameters
• [in] src1:: first range
• [in] src2:: second range
• [in] comp:: object for to compare elements

Exceptions
•

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Sent3, typename Value, typename Compare>
range<Value*> uninit_full_merge (const range<Value*> & 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.

Return range with the elements merged and the size adjusted
Parameters
• [in] dest:: 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
• [in] src1:: first range to merge
• [in] src2:: second range to merge
• [in] comp:: comparison object

template<typename Iter1, typename Sent1, typename Iter2, typename Sent2, typename Iter3, typename Sent3>
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.

Return range with the elements merged and the size adjusted
Parameters
• [in] dest:: 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
• [in] src1:: first range to merge
• [in] src2:: second range to merge
• [in] comp:: comparison object

2.8. API reference
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

Parameters
• [in] src1: : first range to merge
• [in] src2: : second range to merge
• [in] aux: : auxiliary range used in the merge
• [in] comp: : object for to compare elements

Exceptions
•

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

Parameters
• [in] src1: : first range to merge
• [in] src2: : second range to merge
• [in] buf: : auxiliary memory used in the merge
• [in] comp: : object for to compare elements

Exceptions
•

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)
: merge two contiguous buffers

Template Parameters
• Iter: : iterator to the elements
• compare: : object for to compare two elements pointed by Iter iterators

Parameters
• [in] first: : iterator to the first element
• [in] last: : iterator to the element after the last in the range
• [in] comp: : object for to compare elements

Exceptions
•

namespace hpx

namespace ranges
Functions

```cpp
template<typename Iter>
constexpr Iter next (Iter first, typename std::iterator_traits<Iter>::difference_type dist = 1)

template<typename Iter, typename Sent>
constexpr Iter next (Iter first, Sent bound)

template<typename Iter, typename Sent>
constexpr Iter next_ (Iter first, typename std::iterator_traits<Iter>::difference_type n, Sent bound)

template<typename Iter, typename Sent>
constexpr Iter next_ (Iter first, typename std::iterator_traits<Iter>::difference_type n, Sent bound, std::true_type, std::true_type)

namespace hpx

namespace parallel

namespace util

Functions

```cpp
template<typename I1, typename I2>
I2 get_in2_element (util::in_in_result<I1, I2> &&p)

template<typename I1, typename I2>
hpx::future<I2> get_in2_element (hpx::future<util::in_in_result<I1, I2>> &&f)

template<typename I, typename O>
std::pair<I, O> get_pair (util::in_out_result<I, O> &&p)

template<typename I, typename O>
O get_second_element (util::in_out_result<I, O> &&p)

template<typename I, typename O>
hpx::future<std::pair<I, O>> get_pair (hpx::future<util::in_out_result<I, O>> &&f)

template<typename I, typename O>
hpx::future<O> get_second_element (hpx::future<util::in_out_result<I, O>> &&f)

template<typename I, typename O>
hpx::util::iterator_range<I, O> get_subrange (in_out_result<I, O> const &ior)

template<typename I, typename O>
hpx::future<hpx::util::iterator_range<I, O>> get_subrange (hpx::future<in_out_result<I, O>> &&ior)

template<typename I1, typename I2, typename O>
O get_third_element (util::in_out_result<I1, I2, O> &&p)
```
template<typename I1, typename I2, typename O>
hpx::future<O> get_third_element(hpx::future<util::in_in_out_result<I1, I2, O>> &&f)

template<typename ...Ts>
constexpr in_out_out_result<Ts...> make_in_out_out_result(hpx::tuple<Ts...> &&t)

template<typename ...Ts>
hpx::future<in_out_out_result<Ts...>> make_in_out_out_result(hpx::future<hpx::tuple<Ts...>> &&f)

template<typename Iterator, typename Sentinel = Iterator>
util::iterator_range<Iterator, Sentinel> make_subrange(Iterator iterator, Sentinel sentinel)

template<typename Iterator, typename Sentinel = Iterator>
hpx::future<util::iterator_range<Iterator, Sentinel>> make_subrange(hpx::future<Iterator> &&iterator, Sentinel sentinel)

template<typename I, typename F>
struct in_fun_result

Public Functions

template<typename I2, typename F2, typename Enable = typename std::enable_if<std::is_convertible<I const &, I2>::value &&
std::is_convertible<F const &, F2>::value>::type>
constexpr operator in_fun_result<I2, F2>() const &

template<typename I2, typename F2, typename Enable = typename std::enable_if<std::is_convertible<I, I2>::value &&
std::is_convertible<F, F2>::value>::type>
constexpr operator in_fun_result<I2, F2>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)

Public Members

HPX_NO_UNIQUE_ADDRESS I hpx::parallel::util::in_fun_result::in
HPX_NO_UNIQUE_ADDRESS F hpx::parallel::util::in_fun_result::fun

struct in_in_out_result

Public Functions

template<typename III, typename II2, typename O1, typename Enable = typename std::enable_if_t<std::is_convertible<III const &, II2>::value &&
std::is_convertible<O1 const &, O>::value>::type>
constexpr operator in_in_out_result<III, II2, O1>() const &

template<typename II2, typename III, typename O1, typename Enable = typename std::enable_if_t<std::is_convertible<II2, III>::value &&
std::is_convertible<O1, O>::value>::type>
constexpr operator in_in_out_result<II2, III, O1>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)
Public Members

```cpp
HPX_NO_UNIQUE_ADDRESS I1 hpx::parallel::util::in_in_out_result::in1
HPX_NO_UNIQUE_ADDRESS I2 hpx::parallel::util::in_in_out_result::in2
HPX_NO_UNIQUE_ADDRESS O hpx::parallel::util::in_in_out_result::out
```

template<typename I1, typename I2>
struct in_in_result

Public Functions

template<typename III, typename II2, typename Enable = typename std::enable_if<std::is_convertible<II2 const&, III const&>::value && std::is_convertible<II2 const&, III const&>::value>::type>
constexpr operator in_in_result<III, II2>() const &

template<typename III, typename II2, typename Enable = typename std::enable_if<std::is_convertible<III const&, III const&>::value && std::is_convertible<III const&, III const&>::value>::type>
constexpr operator in_in_result<III, II2>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)

Public Members

```cpp
HPX_NO_UNIQUE_ADDRESS I hpx::parallel::util::in_out_out_out_result::in
HPX_NO_UNIQUE_ADDRESS O1 hpx::parallel::util::in_out_out_out_result::out1
HPX_NO_UNIQUE_ADDRESS O2 hpx::parallel::util::in_out_out_out_result::out2
```

template<typename I, typename O1, typename O2>
struct in_out_out_out_result

Public Functions

template<typename II, typename OO1, typename OO2, typename Enable = typename std::enable_if_t<std::is_convertible_v<II const&, II>&& std::is_convertible_v<OO1 const&, II>&& std::is_convertible_v<OO2 const&, II>&&>::type>
constexpr operator in_out_out_out_result<II, OO1, OO2>() const &

template<typename II, typename OO1, typename OO2, typename Enable = typename std::enable_if_t<std::is_convertible_v<II, II>&& std::is_convertible_v<OO1, II>&& std::is_convertible_v<OO2, II>&&>::type>
constexpr operator in_out_out_out_result<II, OO1, OO2>() &&

template<typename Archive>
void serialize(Archive &ar, unsigned)

Public Members

```cpp
HPX_NO_UNIQUE_ADDRESS I hpx::parallel::util::in_out_result::in
HPX_NO_UNIQUE_ADDRESS O1 hpx::parallel::util::in_out_result::out1
HPX_NO_UNIQUE_ADDRESS O2 hpx::parallel::util::in_out_result::out2
```

template<typename I, typename O>
struct in_out_result
Public Functions

template<typename I2, typename O2, typename Enable = typename std::enable_if<!std::is_convertible<const I, I2>::value && std::is_convertible<const O, O2>::value>::type>
constexpr operator in_out_result<I2, O2>() const &

template<typename I2, typename O2, typename Enable = typename std::enable_if<!std::is_convertible<I, I2>::value && std::is_convertible<O, O2>::value>::type>
constexpr operator in_out_result<I2, O2>() & &

template<typename Archive>
void serialize(Archive &ar, unsigned)

Public Members

HPX_NO_UNIQUE_ADDRESS I hpx::parallel::util::in_out_result::in
HPX_NO_UNIQUE_ADDRESS O hpx::parallel::util::in_out_result::out

namespace hpx

namespace util

Functions

template<typename Tag1, typename Tag2, typename T1, typename T2>
hpx::future<tagged_pair<Tag1(typename std::decay<T1>::type), Tag2>
  typename std::decay<T2>::type> make_tagged_pair(hpx::future<std::pair<T1, T2>> & &f

template<typename Tag1, typename Tag2, typename ...Ts>
hpx::future<tagged_pair<Tag1(hpx::tuple_element<0, hpx::tuple<Ts...>::type), Tag2
  hpx::tuple_element<1, hpx::tuple<Ts...>::type>> make_tagged_pair(hpx::future<hpx::tuple<Ts...>> & &f

namespace hpx

namespace util

Functions

template<typename ...Tags, typename ...Ts>
hpx::future<typename detail::tagged_tuple_helper<hpx::tuple<Ts...>, typename util::make_index_pack<sizeof...(Tags)>>

namespace hpx

namespace parallel

namespace util
Functions

```cpp
template<typename InIter, typename Sent, typename OutIter>
constexpr in_out_result<InIter, OutIter> copy(InIter first, Sent last, OutIter dest)

template<typename InIter, typename OutIter>
constexpr void copy_synchronize(InIter const &first, OutIter const &dest)

template<typename InIter, typename Sent, typename OutIter>
constexpr in_out_result<InIter, OutIter> move(InIter first, Sent last, OutIter dest)

template<typename InIter, typename OutIter>
constexpr in_out_result<InIter, OutIter> move_n(InIter first, std::size_t count, OutIter dest)
```

Variables

```cpp
template<typename ExPolicy> HPX_INLINE_CONSTEXPR_VARIABLE copy_n_t<ExPolicy> hpx::parallel::util::copy_n=copy_n_t<ExPolicy>{};

template<typename ExPolicy>
struct copy_n_t : public hpx::functional::detail::tag_fallback<copy_n_t<ExPolicy>>;
```

Friends

```cpp
template<typename InIter, typename OutIter>
friend constexpr in_out_result<InIter, OutIter> tag_fallback_invoke(hpx::parallel::util::copy_n_t<ExPolicy>, InIter first, std::size_t count, OutIter dest);
```

```cpp
namespace hpx

namespace parallel

namespace util

Variables

HPXINLINE_CONSTEXPR_VARIABLE transform_loop_t hpx::parallel::util::transform_loop=transform_loop_t{};
HPXINLINE_CONSTEXPR_VARIABLE transform_loop_ind_t hpx::parallel::util::transform_loop_ind=transform_loop_ind_t{};
HPXINLINE_CONSTEXPR_VARIABLE transform_binary_loop_t<ExPolicy> hpx::parallel::util::transform_binary_loop=transform_binary_loop_t<ExPolicy>{};
HPXINLINE_CONSTEXPR_VARIABLE transform_binary_loop_ind_t<ExPolicy> hpx::parallel::util::transform_binary_loop_ind=transform_binary_loop_ind_t<ExPolicy>{};
HPXINLINE_CONSTEXPR_VARIABLE transform_loop_n_t<ExPolicy> hpx::parallel::util::transform_loop_n=transform_loop_n_t<ExPolicy>{};
HPXINLINE_CONSTEXPR_VARIABLE transform_loop_n_ind_t<ExPolicy> hpx::parallel::util::transform_loop_n_ind=transform_loop_n_ind_t<ExPolicy>{};
HPXINLINE_CONSTEXPR_VARIABLE transform_binary_loop_n_t<ExPolicy> hpx::parallel::util::transform_binary_loop_n=transform_binary_loop_n_t<ExPolicy>{};
```
template<typename ExPolicy> HPX_INLINE_CONSTEXPR_VARIABLE transform_binary_loop_ind_n_t<ExPolicy> hpx::parallel::util::transform_binary_loop_ind_n_t = transform_binary_loop_ind_n_t<ExPolicy>{}

struct transform_binary_loop_ind_n_t: public hpx::functional::detail::tag_fallback<transform_binary_loop_ind_n_t<ExPolicy> >

Friends

template<typename InIter1, typename InIter2, typename OutIter, typename F>
friend constexpr hpx::tuple<InIter1, InIter2, OutIter> tag_fallback_invoke (hpx::parallel::util::transform_binary_loop_ind_n_t<ExPolicy>, InIter1 first1, std::size_t count, InIter2 first2, OutIter dest, F &&f)

Chapter 2. What's so special about HPX?
friend constexpr
util::in_in_out_result<InIter1B, InIter2B, OutIter>
tag_fallback_invoke
(hpx::parallel::util::transform_binary_loop_n_t<ExPolicy>,
InIter1 first1, InIter1E last1,
InIter2B first2, InIter2E last2, OutIter dest, F &&f)

template<typename ExPolicy>
struct transform_binary_loop_n_t : public hpx::functional::detail::tag_fallback
(transform_binary_loop_n_t<ExPolicy>)

Friends

template<typename InIter1, typename InIter2, typename OutIter, typename F>
friend constexpr
hpx::tuple<InIter1, InIter2, OutIter>
tag_fallback_invoke
(hpx::parallel::util::transform_binary_loop_t<ExPolicy>,
InIter1 first1, std::size_t count, InIter2 first2, OutIter dest, F &&f)

template<typename ExPolicy>
struct transform_binary_loop_t : public hpx::functional::detail::tag_fallback
(transform_binary_loop_t<ExPolicy>)

Friends

template<typename InIter1B, typename InIter1E, typename InIter2, typename OutIter, typename F>
friend constexpr
util::in_in_out_result<InIter1B, InIter2, OutIter>
tag_fallback_invoke
(hpx::parallel::util::transform_binary_loop_t<ExPolicy>,
InIter1B first1, InIter1E last1, InIter2 first2, OutIter dest, F &&f)
template<typename InIter1B, typename InIter1E, typename InIter2B, typename InIter2E, typename OutIter, typename F>
friend constexpr util::in_in_out_result<InIter1B, InIter2B, OutIter> tag_fallback_invoke(hpx::parallel::util::transform_binary_loop_t<ExPolicy>, InIter1B first1, InIter1E last1, InIter2B first2, InIter2E last2, OutIter dest, F &&f)

struct transform_loop_ind_t : public hpx::functional::detail::tag_fallback<transform_loop_ind_t>

Friends

template<typename ExPolicy, typename IterB, typename IterE, typename OutIter, typename F>
friend constexpr util::in_out_result<IterB, OutIter> tag_fallback_invoke(hpx::parallel::util::transform_loop_ind_t, ExPolicy &&, IterB it, IterE end, OutIter dest, F &&f)

template<typename ExPolicy>
struct transform_loop_n_ind_t : public hpx::functional::detail::tag_fallback<transform_loop_n_ind_t<ExPolicy>>

Friends

template<typename Iter, typename OutIter, typename F>
friend constexpr std::pair<Iter, OutIter> tag_fallback_invoke(hpx::parallel::util::transform_loop_n_ind_t, Iter it, std::size_t count, OutIter dest, F &&f)

template<typename ExPolicy>
struct transform_loop_n_t : public hpx::functional::detail::tag_fallback<transform_loop_n_t<ExPolicy>>
Friends

template<
type
ame Iter, typename OutIter, 
type
name F>
friend constexpr std::pair<
Iter, OutIter> tag_fallback_invoke (hpx::parallel::util::transform_loop_n_t<
Iter>
It, 
std::size_t

count, 
OutIter

dest, F &&f)

struct transform_loop_t : public hpx::functional::detail::tag_fallback<transform_loop_t>

Friends

template<
type
ame ExPolicy, typename IterB, typename IterE, typename OutIter, 
type
name F>
friend constexpr util::in_out_result<
IterB

, OutIter

> tag_fallback_invoke (hpx::parallel::util::transform_loop_t<
ExPolicy>&&, 
IterB

it, 
IterE

end, 
OutIter

dest,
F

&&f)

allocator_support

The contents of this module can be included with the header hpx/modules/allocator_support.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/allocator_support.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Functions

void *__aligned_alloc (std::size_t alignment, std::size_t size)

void __aligned_free (void *)

namespace hpx

namespace util
Functions

template<typename T>
constexpr bool operator==(aligned_allocator<T> const&, aligned_allocator<T> const&)

template<typename T>
constexpr bool operator!=(aligned_allocator<T> const&, aligned_allocator<T> const&)

template<typename T = int>
struct aligned_allocator

Public Types

typedef T value_type

typedef T* pointer

typedef T& reference

typedef T const& const_reference

typedef std::size_t size_type

typedef std::ptrdiff_t difference_type

typedef std::true_type is_always_equal

typedef std::true_type propagate_on_container_move_assignment

Public Functions

aligned_allocator()

template<typename U>
aligned_allocator(aligned_allocator<U> const&)

pointer address(reference x) const

cost_pointer address(const_reference x) const

HPX_NODISCARD pointer hpx::util::aligned_allocator::allocate(size_type n, void const*)

void deallocate(pointer p, size_type)

size_type max_size() const

template<typename U, typename ...Args>
void construct(U* p, Args&&... args)

template<typename U>
void destroy(U* p)
Public Members

const typedef T* hpx::util::aligned_allocator::const_pointer

template<typename U>
struct rebind

Public Types

template<>
typedef aligned_allocator<U> other

namespace hpx

namespace util

template<typename Allocator>
struct allocator_deleter

Public Functions

template<typename SharedState>
void operator() (SharedState *state)

Public Members

Allocator alloc_

namespace hpx

namespace util

Typedefs

template<typename T = int>
using internal_allocator = std::allocator<T>

namespace hpx

namespace traits
Variables

template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_allocator_v = is_allocator<T>::value

asio

The contents of this module can be included with the header hpx/modules/asio.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/asio.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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)

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)

std::string resolve_public_ip_address ()

std::string cleanup_ip_address (std::string const &addr)

domain_endpoint_iterator_type connect_begin (std::string const &address, std::uint16_t port, asio::io_context &io_service)

template<typename Locality>

domain_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.

domain_endpoint_iterator_type connect_end ()

domain_endpoint_iterator_type accept_begin (std::string const &address, std::uint16_t port, asio::io_context &io_service)

template<typename Locality>

domain_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.

domain_endpoint_iterator_type accept_end ()

bool split_ip_address (std::string const &v, std::string &host, std::uint16_t &port)
namespace hpx

namespace util

struct map_hostnames

Public Types

typedef util::function_nonser<std::string (std::string const &)> transform_function_type

Public Functions

map_hostnames (bool debug = false)
void use_suffix (std::string const &suffix)
void use_prefix (std::string const &prefix)
void use_transform (transform_function_type const &f)
std::string map (std::string host_name, std::uint16_t port) const

Private Members

transform_function_type transform_
std::string suffix_
std::string prefix_
bool debug_

assertion

The contents of this module can be included with the header hpx/modules/assertion.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/assertion.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

HPX_ASSERT_CURRENT_FUNCTION

namespace hpx

namespace assertion
### Functions

```cpp
std::ostream &operator<<(std::ostream &os, source_location const &loc)
```

#### struct source_location

```cpp
#include <source_location.hpp>
```

This contains the location information where `HPX_ASSERT` has been called.

#### Public Members

```cpp
const char *file_name
unsigned line_number
const char *function_name
```

### 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()`.

**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`

 asserts are enabled if `HPX_DEBUG` is set. This is the default for `CMAKE_BUILD_TYPE=Debug`

**HPX_ASSERT_MSG** *(expr, msg)*

See **HPX_ASSERT**

### Typedefs

```cpp
using assertion_handler = void (*)(source_location const &loc, const char *expr, std::string const &msg)
```

The signature for an assertion handler.
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

The contents of this module can be included with the header `hpx/modules/async_base.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/async_base.hpp`, not the particular header in which the functionality you would like to use is defined. See **Public API** for a list of names that are part of the public HPX API.

```cpp
namespace hpx
{

    // Functions

template<typename F, typename ...Ts>
bool apply (F &&f, Ts&&... ts)

namespace hpx
{

    // Functions

template<typename F, typename ...Ts>
decltype(auto) async (F &&f, Ts&&... ts)

namespace hpx
{

    // Functions

template<typename F, typename ...Ts>
auto dataflow (F &&f, Ts&&... ts)

    template<typename Allocator, typename F, typename ...Ts>
auto dataflow_alloc (Allocator const &alloc, F &&f, Ts&&... ts)

namespace hpx
{

    struct launch : public detail::policy_holder<>
    {
        #include <launch_policy.hpp> Launch policies for hpx::async etc.
    }

```
Public Functions

```cpp
constexpr launch ()
Default constructor. This creates a launch policy representing all possible launch modes.
```

```cpp
constexpr launch (detail::async_policy p)
Create a launch policy representing asynchronous execution.
```

```cpp
constexpr launch (detail::fork_policy p)
Create a launch policy representing asynchronous execution. The new thread is executed in a preferred way.
```

```cpp
constexpr launch (detail::sync_policy p)
Create a launch policy representing synchronous execution.
```

```cpp
constexpr launch (detail::deferred_policy p)
Create a launch policy representing deferred execution.
```

```cpp
constexpr launch (detail::apply_policy p)
Create a launch policy representing fire and forget execution.
```

```cpp
template<typename F>
constexpr launch (detail::select_policy<F> const &p)
Create a launch policy representing fire and forget execution.
```

```cpp
template<typename Launch, typename Enable = std::enable_if_t<hpx::traits::is_launch_policy_v<Launch>>>
constexpr launch (Launch l, threads::thread_priority priority, threads::thread_stacksize stacksize, threads::thread_schedule_hint hint)
```

Public Static Attributes

```cpp
const detail::async_policy async
Predefined launch policy representing asynchronous execution.
```

```cpp
const detail::fork_policy fork
Predefined launch policy representing asynchronous execution. The new thread is executed in a preferred way.
```

```cpp
const detail::sync_policy sync
Predefined launch policy representing synchronous execution.
```

```cpp
const detail::deferred_policy deferred
Predefined launch policy representing deferred execution.
```

```cpp
const detail::apply_policy apply
Predefined launch policy representing fire and forget execution.
```

```cpp
const detail::select_policy_generator select
Predefined launch policy representing delayed policy selection.
```
Friends

```cpp
launch tag_invoke (hpx::execution::experimental::with_priority_t, launch const &policy,
threads::thread_priority priority)

friend constexpr hpx::threads::thread_priority tag_invoke (hpx::execution::experimental::get_priority_t,
launch const &policy)

launch tag_invoke (hpx::execution::experimental::with_stacksize_t, launch const &policy,
threads::thread_stacksize stacksize)

friend constexpr hpx::threads::thread_stacksize tag_invoke (hpx::execution::experimental::get_stacksize_t,
launch const &policy)

launch tag_invoke (hpx::execution::experimental::with_hint_t, launch const &policy,
threads::thread_schedule_hint hint)

friend constexpr hpx::threads::thread_schedule_hint tag_invoke (hpx::execution::experimental::get_hint_t,
launch const &policy)
```

namespace hpx

namespace execution

namespace experimental

Variables

```cpp
hpx::execution::experimental::with_priority_t with_priority
hpx::execution::experimental::get_priority_t get_priority
hpx::execution::experimental::with_stacksize_t with_stacksize
hpx::execution::experimental::get_stacksize_t get_stacksize
hpx::execution::experimental::with_hint_t with_hint
hpx::execution::experimental::get_hint_t get_hint
hpx::execution::experimental::with_annotation_t with_annotation
hpx::execution::experimental::get_annotation_t get_annotation
```

namespace hpx

Functions

```cpp
template<typename F, typename ...Ts>
auto sync (F &&f, Ts&... ts)
```

namespace hpx

namespace traits
Variables

```cpp
template<typename Policy>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_launch_policy_v = is_launch_policy<Policy>::value
```

async_combinators

The contents of this module can be included with the header `hpx/modules/async_combinators.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/async_combinators.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
```

```cpp
namespace lcos
```

### Functions

```cpp
template<typename Future, typename F>
std::enable_if<!std::is_void<typename traits::future_traits<Future>::type>::value, std::size_t>::type
wait(Future &&f1, F &&f)
```

The one argument version is special in the sense that it returns the expected value directly (without wrapping it into a tuple).

```cpp
template<typename Future, typename F>
std::enable_if<!std::is_void<typename traits::future_traits<Future>::type>::value, std::size_t>::type
wait(Future &&f1, F &&f)
```

```cpp
template<typename Future, typename F>
std::size_t
wait(std::vector<Future> &lazy_values, F &&f, std::int32_t = 10)
```

```cpp
template<typename Future, typename F>
std::size_t
wait(std::vector<Future> &&lazy_values, F &&f, std::int32_t = 10)
```

```cpp
template<typename Future, typename F>
std::size_t
wait(std::vector<Future> const &lazy_values, F &&f, std::int32_t = 10)
```

```
namespace hpx
```

### Functions

```cpp
template<typename ...Ts>
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`. 
Return 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.

Note The following cases are special:

```cpp
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>`

```cpp
template<typename T>
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`.

Return 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.

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.

```cpp
namespace hpx

Functions

```cpp
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. Exceptional futures will not cause `wait_all` to throw an exception.

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.

```cpp
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. Exceptional futures will not cause wait_all to throw an exception.

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>
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. Exceptional futures will not cause wait_all to throw an exception.

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 (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. Exceptional futures will not cause wait_all to throw an exception.

Parameters

• futures: An arbitrary number of future or shared_future objects, possibly holding different types for which wait_all should wait.

template<typename InputIter>
InputIter 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.

Return The function wait_all_n will return an iterator referring to the first element in the input sequence after the last processed element.

Note The function wait_all_n returns after all futures have become ready. All input futures are still valid after wait_all_n returns. Exceptional futures will not cause wait_all to throw an exception.

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.

namespace hpx
Functions

```cpp
template<typename InputIter>
void wait_any (InputIter first, InputIter last, error_code &ec = throws)
```

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** 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**

- `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.
- `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.

```cpp
template<typename R>
void wait_any (std::vector<future<R>> &futures, error_code &ec = throws)
```

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** 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**

- `futures`: [in] A vector holding an arbitrary amount of `future` or `shared_future` objects for which `wait_any` 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.

```cpp
template<typename R, std::size_t N>void hpx::wait_any (std::array< future<R>, N > & futures, error_code &ec = throws)
```

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** 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

• futures: [in] An arbitrary number of future or shared_future objects, possibly holding different types for which \texttt{wait\_any} should wait.

• ec: [in,out] this represents the error status on exit, if this is pre-initialized to \texttt{hpx::throws} the function will throw on error instead.

\begin{verbatim}
template<typename ...T>
void wait\_any (error\_code &ec, T&... futures)
\end{verbatim}

The function \texttt{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.

\begin{itemize}
\item \textbf{Note} The function \texttt{wait\_any} returns after at least one future has become ready. All input futures are still valid after \texttt{wait\_any} returns.
\item \textbf{Note} As long as \texttt{ec} is not pre-initialized to \texttt{hpx::throws} this function doesn’t throw but returns the result code using the parameter \texttt{ec}. Otherwise it throws an instance of \texttt{hpx::exception}.
\item \textbf{Note} None of the futures in the input sequence are invalidated.
\end{itemize}

Parameters

• futures: [in] An arbitrary number of future or shared_future objects, possibly holding different types for which \texttt{wait\_any} should wait.

• ec: [in,out] this represents the error status on exit, if this is pre-initialized to \texttt{hpx::throws} the function will throw on error instead.

\begin{verbatim}
template<typename ...T>
void wait\_any (T&... futures)
\end{verbatim}

The function \texttt{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.

\begin{itemize}
\item \textbf{Note} The function \texttt{wait\_any} returns after at least one future has become ready. All input futures are still valid after \texttt{wait\_any} returns.
\item \textbf{Note} None of the futures in the input sequence are invalidated.
\end{itemize}

Parameters

\begin{verbatim}
template<typename InputIter>
InputIter wait\_any\_n (InputIter first, std::size_t count, error\_code &ec = throws)
\end{verbatim}

The function \texttt{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.

\begin{itemize}
\item \textbf{Note} The function \texttt{wait\_any\_n} returns after at least one future has become ready. All input futures are still valid after \texttt{wait\_any\_n} returns.
\item \textbf{Return} The function \texttt{wait\_all\_n} will return an iterator referring to the first element in the input sequence after the last processed element.
\item \textbf{Note} As long as \texttt{ec} is not pre-initialized to \texttt{hpx::throws} this function doesn’t throw but returns the result code using the parameter \texttt{ec}. Otherwise it throws an instance of \texttt{hpx::exception}.
\end{itemize}
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 `wait_any_n` should wait.
- **count**: [in] The number of elements in the sequence starting at `first`.
- **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.

```cpp
namespace hpx

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.

```cpp
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.

namespace hpx
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 future returned by the function `wait_some` becomes ready when at least \( n \) argument futures have become ready.

**Return** Returns a future holding the same list of futures as has been passed to `wait_some`.

- `future<vector<future<T>>>`: If the input cardinality is unknown at compile time and the futures are all of the same type.

**Note** Calling this version of `wait_some` where first == last, returns a future with an empty vector 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 `wait_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.
- `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.

The function `wait_all` returns after \( n \) futures have become ready. All input futures are still valid after `wait_all` returns.

**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 `wait_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.
- `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.
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.

**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.
- **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.

**Note** The function `wait_all` returns after n futures have become ready. All input futures are still valid after `wait_all` returns.

**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 `wait_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 `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.
InputIter wait_some_n (std::size_t n, Iterator first, std::size_t count, error_code &ec = throws)

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_all returns after n futures have become ready. All input futures are still valid after wait_all returns.

Return This function returns an Iterator referring to the first element after the last processed input element.

Note Calling this version of wait_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 vector 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 wait_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.
- 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.

namespace hpx

Functions

template<typename InputIter, typename Container = vector<future<typename std::iterator_traits<InputIter>::value_type>>> future<Container<future<R>>> when_all (InputIter first, InputIter last)

The function when_all 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 they finished executing.

Return 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.

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.
template<typename Range>
future<Range> when_all (Range &values)

The function when_all 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 they finished executing.

Return 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.

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.

template<typename ... T>
future<tuple<future<T>...>> when_all (T&... futures)

The function when_all 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 they finished executing.

Return 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.

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.

template<typename InputIter, typename Container = vector<future<typename std::iterator_traits<InputIter>::value_type>>> future<Container> when_all_n (InputIter begin, std::size_t count)

The function when_all_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 they finished executing.

Return 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.
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.

Exceptions

• 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.

namespace hpx

Functions

template<
typeinfo InputIter, typename Container = vector<
typeinfo future<std::iterator_traits<InputIter>::value_type>>
future<when_any_result<
Container>> when_any (InputIter first, InputIter last)

The function when_any is a non-deterministic choice operator. It OR-composes all future objects given and returns a new future object representing the same list of futures after one future of that list finishes execution.

Return 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.

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.

template<
typeinfo Range>
future<when_any_result<
Range>> when_any (Range &values)

The function when_any is a non-deterministic choice operator. It OR-composes all future objects given and returns a new future object representing the same list of futures after one future of that list finishes execution.

Return 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.

Parameters
• **values**: [in] A range holding an arbitrary amount of `future` or `shared_future` objects for which `when_any` should wait.

```cpp
template<typename ...T>
future<when_any_result<tuple<future<T>...>>> when_any(T&&... futures)
```

The function `when_any` is a non-deterministic choice operator. It OR-composes all future objects given and returns a new future object representing the same list of futures after one future of that list finishes execution.

**Return** 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.

**Parameters**

- `futures`: [in] An arbitrary number of `future` or `shared_future` objects, possibly holding different types for which `when_any` should wait.

```cpp
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)
```

The function `when_any_n` is a non-deterministic choice operator. It OR-composes all future objects given and returns a new future object representing the same list of futures after one future of that list finishes execution.

**Return** 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.

**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`.

```cpp
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.
```

```cpp
namespace hpx

Functions

```cpp
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.

**Return** Returns a future representing the event of all input futures being ready.

**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.

```cpp
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.

**Return** Returns a future representing the event of all input futures being ready.

**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 ... 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.

**Return** Returns a future representing the event of all input futures being ready.

**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>
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.

**Return** Returns a future holding the iterator pointing to the first element after the last one.

**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.

---

namespace hpx

---

**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, error_code &ec = throws)

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.

Return 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.

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.
- `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 Range>
future<when_some_result<Range>> when_some (std::size_t n, Range &&futures, error_code &ec = throws)
```

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.

Return 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.

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.
- `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 ...T>
future<when_some_result<tuple<future<T>...>>> when_some (std::size_t n, error_code &ec, T&&... 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.

Return 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.

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.
- 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.
- futures: [in] An arbitrary number of future or shared_future objects, possibly holding different types for which when_some should wait.

template<typename ...T>
future<when_some_result<tuple<future<T>...>>> when_some (std::size_t n, T&&... 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.

Return 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.

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.

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.

The future returned by the function when_some_n becomes ready when at least n argument futures
have become ready.

Returns a when_some_result holding the same list of futures as has been passed to when_some
and indices pointing to ready futures.

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) fu-
ture, 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.

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.
async_cuda

The contents of this module can be included with the header `hpx/modules/async_cuda.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/async_cuda.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

    namespace cuda

        namespace experimental

            struct cuda_event_pool

                Public Functions

                cuda_event_pool()
                ~cuda_event_pool()
                bool pop (cudaEvent_t &event)
                bool push (cudaEvent_t event)

                Public Static Functions

                static cuda_event_pool &get_event_pool()

                Public Static Attributes

                constexpr int initial_events_in_pool = 128

                Private Functions

                void add_event_to_pool()

                Private Members

                boost::lockfree::stack<cudaEvent_t, boost::lockfree::fixed_sized<false>> free_list_

        namespace cuda

            namespace experimental
Functions

```cpp
void check_cuda_error(cudaError_t err)
```

```cpp
struct cuda_exception : public exception
```

Public Functions

```cpp
cuda_exception(const std::string &msg, cudaError_t err)
cudaError_t get_cuda_errorcode()
```

Protected Attributes

```cpp
cudaError_t err_
```

namespace hpx

```cpp
namespace cuda
```

```cpp
namespace experimental
```

```cpp
struct cuda_executor : public hpx::cuda::experimental::cuda_executor_base
```

Public Functions

```cpp
cuda_executor (std::size_t device, bool event_mode = true)
```

```cpp
~cuda_executor ()
```

```cpp
template<typename F, typename ...Ts>
decay(auto) post (F &&f, Ts&&... ts)
```

```cpp
template<typename F, typename ...Ts>
decay(auto) async_execute (F &&f, Ts&&... ts)
```

Protected Functions

```cpp
template<typename R, typename ...Params, typename ...Args>
void apply (R (*cuda_function)) Params... .Args&&... args
```

```cpp
template<typename R, typename ...Params, typename ...Args>
hpx::future<void> async (R (*cuda_kernel)) Params... .Args&&... args
```

```cpp
struct cuda_executor_base
```

```cpp
Subclassed by hpx::cuda::experimental::cuda_executor
```
Public Types

```cpp
using future_type = hpx::future<void>
```

Public Functions

```cpp
cuda_executor_base (std::size_t device, bool event_mode)
future_type get_future()
```

Protected Attributes

```cpp
int device_
bool event_mode_
cudaStream_t stream_
std::shared_ptr<hpx::cuda::experimental::target> target_
```

namespace hpx
	namespace cuda
	namespace experimental

Typedefs

```cpp
using event_mode = std::true_type
using callback_mode = std::false_type
```

namespace hpx
	namespace cuda
	namespace experimental

struct enable_user_polling

Public Functions

```cpp
enable_user_polling (std::string const &pool_name = "")
~enable_user_polling()
```
**Private Members**

`std::string pool_name_`

```cpp
namespace hpx

namespace cuda

namespace experimental

**Functions**

`std::vector<target> get_local_targets()`

`void print_local_targets()`
```

namespace hpx

namespace compute

namespace cuda

**Typedefs**

`using instead = hpx::cuda::experimental::target`

namespace cuda

namespace experimental

**Functions**

`target & get_default_target()`

```cpp
struct target

**Public Functions**

`target()`

`target(int device)`

`target(target const &rhs)`

`target(target &&rhs)`

`target & operator=(target const &rhs)`

`target & operator=(target &&rhs)`
```
native_handle_type &native_handle() const

void synchronize() const

hpX::future<void> get_future_with_event() const

hpX::future<void> get_future_with_callback() const

template<typename Allocator>
hpX::future<void> get_future_with_event(Allocator const &alloc) const

template<typename Allocator>
hpX::future<void> get_future_with_callback(Allocator const &alloc) const

Public Static Functions

static std::vector<target> get_local_targets()

Private Members

native_handle_type handle_

Friends

bool operator==(target const &lhs, target const &rhs)

struct native_handle_type

Public Types

typedef hpx::lcos::local::spinlock mutex_type

Public Functions

native_handle_type(int device = 0)

~native_handle_type()

native_handle_type(native_handle_type const &rhs)

native_handle_type(native_handle_type &&rhs)

native_handle_type &operator=(native_handle_type const &rhs)

native_handle_type &operator=(native_handle_type &&rhs)

cudaStream_t get_stream() const

int get_device() const

std::size_t processing_units() const
std::size_t processor_family() const

std::string processor_name() const

void reset()

**Private Functions**

void init_processing_units()

**Private Members**

mutex_type mtx_

int device_

std::size_t processing_units_

std::size_t processor_family_

std::string processor_name_

cudaStream_t stream_

**Friends**

friend hpx::cuda::experimental::target

namespace hpx

namespace cuda

namespace experimental

**Variables**

hpx::cuda::experimental::transform_stream_t transform_stream

struct transform_stream_t: public hpx::functional::detail::tag_fallback<transform_stream_t>

**Friends**

template<
typename S, typename F, typename = std::enable_if_t<!std::is_same<std::decay_t<F>, cudaStream_t>::value>>
friend constexpr auto tag_fallback_invoke (transform_stream_t, S &&s, F &&f, cudaStream_t stream = { })

template<
typename F>
friend constexpr auto tag_fallback_invoke (transform_stream_t, F &&f, cudaStream_t stream = { })
async_local

The contents of this module can be included with the header `hpx/modules/async_local.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/async_local.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
{
    namespace Functions
    {
        template<typename Action, typename F, typename ...Ts>
        auto async(F &&f, Ts &&...ts)
    }
}
```

async_mpi

The contents of this module can be included with the header `hpx/modules/async_mpi.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/async_mpi.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
{
    namespace mpi
    {
        namespace experimental
        {
            struct executor
            {
                using execution_category = hpx::execution::parallel_execution_tag
                using executor_parameters_type = hpx::execution::static_chunk_size
            }
        }
    }
}
```
Public Functions

**constexpr executor** (MPI_Comm `communicator = MPI_COMM_WORLD`)

```cpp
template<typename F, typename ...Ts>
decltype(auto) async_execute (F &&f, Ts&&... ts) const
```

```cpp
std::size_t in_flight_estimate () const
```

Private Members

```cpp
MPI_Comm communicator_
```

```cpp
namespace hpx
```

```cpp
namespace mpi
```

```cpp
namespace experimental
```

Typedefs

```cpp
using print_on = debug::enable_print<false>
```

Functions

```cpp
static constexpr print_on hpx::mpi::experimental::mpi_debug("MPI_FUT")
```

```cpp
void set_error_handler ()
```

```cpp
hpx::future<void> get_future (MPI_Request request)
```

```cpp
hpx::threads::policies::detail::polling_status poll ()
```

```cpp
void wait ()
```

```cpp
template<typename F>
void wait (F &&f)
```

```cpp
void init (bool init_mpi = false, std::string const &pool_name = "", bool init_errorhandler = false)
```

```cpp
void finalize (std::string const &pool_name = "")
```

```cpp
template<typename ...Args>
void debug (Args&&... args)
```

struct enable_user_polling
Public Functions

enable_user_polling(const std::string &pool_name = "")
~enable_user_polling()

Private Members

std::string pool_name_

batch_environments

The contents of this module can be included with the header hpx/modules/batch_environments.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/batch_environments.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace util

namespace batch_environments

struct alps_environment

Public Functions

alps_environment(const std::vector<std::string> &nodelist, bool debug)
bool valid() const
std::size_t node_num() const
std::size_t num_threads() const
std::size_t num_localities() const

Private Members

std::size_t node_num_
std::size_t num_threads_
std::size_t num_localities_
bool valid_

namespace hpx
namespace util

struct batch_environment

Public Types

typedef std::map<asio::ip::tcp::endpoint, std::pair<std::string, std::size_t>> node_map_type

Public Functions

batch_environment (std::vector<std::string> &nodelist, bool have_mpi = false, bool debug = false, bool enable = true)
std::string init_from_nodelist (std::vector<std::string> const &nodes, std::string const &agas_host)
std::size_t retrieve_number_of_threads () const
std::size_t retrieve_number_of_localities () const
std::size_t retrieve_node_number () const
std::string host_name () const
std::string host_name (std::string const &def_hpx_name) const
std::string agas_host_name (std::string const &def_agas) const
std::size_t agas_node () const
bool found_batch_environment () const
std::string get_batch_name () const

Public Members

std::string agas_node_
std::size_t agas_node_num_
std::size_t node_num_
std::size_t num_threads_
nodes_map_type nodes_
std::size_t num_localities_
std::string batch_name_
bool debug_

namespace hpx

namespace util
namespace batch_environments

struct pbs_environment

Public Functions

pbs_environment (std::vector<std::string> &nodelist, bool have mpi, bool debug)

bool valid() const
std::size_t node_num() const
std::size_t num_threads() const
std::size_t num_localities() const

Private Functions

void read_nodefile (std::vector<std::string> &nodelist, bool have mpi, bool debug)
void read_nodelist (std::vector<std::string> &nodelist, bool debug)

Private Members

std::size_t node_num_
std::size_t num_localities_
std::size_t num_threads_
bool valid_

namespace hpx

namespace util

namespace batch_environments

struct slurm_environment

Public Functions

slurm_environment (std::vector<std::string> &nodelist, bool debug)

bool valid() const
std::size_t node_num() const
std::size_t num_threads() const
std::size_t num_localities() const
Private Functions

void retrieve_number_of_localities (bool debug)
void retrieve_number_of_tasks (bool debug)
void retrieve_nodelist (std::vector<std::string> &nodes, bool debug)
void retrieve_number_of_threads ()

Private Members

std::size_t node_num_
std::size_t num_threads_
std::size_t num_tasks_
std::size_t num_localities_
bool valid_

cache

The contents of this module can be included with the header hpx/modules/cache.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/cache.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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

```cpp
typedef Key key_type
typedef Entry entry_type
typedef UpdatePolicy update_policy_type
typedef InsertPolicy insert_policy_type
typedef CacheStorage storage_type
typedef Statistics statistics_type
typedef entry_type::value_type value_type
typedef storage_type::size_type size_type
typedef storage_type::value_type storage_value_type
```

### Public Functions

```cpp
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.

```cpp
local_cache(local_cache &&other)
```

**Return** The current size of this cache instance.

```cpp
size_type size() const
```

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`.

Return 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.

bool `reserve` (size_type max_size)
Change the maximum size this cache can grow to.

Return 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.

Parameters
- `max_size`: [in] The new maximum size this cache will be allowed to grow to.

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.

Return This function returns `true` if the cache holds the referenced entry, otherwise it returns `false`.

Parameters
- `k`: [in] The key for the entry which should be looked up in the cache.

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.

Return This function returns `true` if the cache holds the referenced entry, otherwise it returns `false`.

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.

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.

Return This function returns `true` if the cache holds the referenced entry, otherwise it returns `false`.

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.

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.
Return This function returns \textit{true} if the cache holds the referenced entry, otherwise it returns \textit{false}.

Parameters
- \textit{k}: \textit{[in]} The key for the entry which should be retrieved from the cache
- \textit{val}: \textit{[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.

bool \texttt{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 \texttt{entry::insert} of the newly constructed entry instance. If either of these functions returns \textit{false} the key/value pair doesn’t get inserted into the cache and the \texttt{insert} function will return \textit{false}. Other reasons for this function to fail (return \textit{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.

Return This function returns \textit{true} if the entry has been successfully added to the cache, otherwise it returns \textit{false}.

Parameters
- \textit{k}: \textit{[in]} The key for the entry which should be added to the cache.
- \textit{value}: \textit{[in]} The value which should be added to the cache.

bool \texttt{insert (key\_type const \&k, entry\_type \&e)}
Insert a new entry into this cache.

Note This function invokes both, the insert policy as provided to the constructor and the function \texttt{entry::insert} of the provided entry instance. If either of these functions returns \textit{false} the key/value pair doesn’t get inserted into the cache and the \texttt{insert} function will return \textit{false}. Other reasons for this function to fail (return \textit{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.

Return This function returns \textit{true} if the entry has been successfully added to the cache, otherwise it returns \textit{false}.

Parameters
- \textit{k}: \textit{[in]} The key for the entry which should be added to the cache.
- \textit{value}: \textit{[in]} The entry which should be added to the cache.

bool \texttt{update (key\_type const \&k, value\_type const \&val)}
Update an existing element in this cache.

Note The function will call the entry’s \texttt{entry::touch} function if the indexed value is found in the cache.

Note The difference to the other overload of the \texttt{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.

Return This function returns \textit{true} if the entry has been successfully updated, otherwise it returns \textit{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.

Parameters
- \textit{k}: \textit{[in]} The key for the value which should be updated in the cache.
- \textit{value}: \textit{[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.

\texttt{template<typename F>
bool update_if(key_type const &k, value_type const &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.
Return 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.
Parameters
• k: [in] The key for the value which should be updated in the cache.
• value: [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.

bool update(key_type const &k, entry_type &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 replaces the cached value only, leaving the
cache entry properties untouched.
Return 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.
Parameters
• k: [in] The key for the entry which should be updated in the cache.
• value: [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.

template<typename Func>
size_type erase(Func const &ep = policies::always<storage_value_type>())
Remove stored entries from the cache for which the supplied function object returns true.

Return 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).
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.

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.

**Return** 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).

```cpp
void clear()
Clear the cache.

Unconditionally removes all stored entries from the cache.
```

```cpp
statistics_type const &get_statistics() const
Allow to access the embedded statistics instance.

**Return** This function returns a reference to the statistics instance embedded inside this cache
```

```cpp
statistics_type &get_statistics()
```

**Protected Functions**

```cpp
bool free_space(long num_free)
```

**Private Types**

```cpp
typedef storage_type::iterator iterator
typedef storage_type::const_iterator const_iterator
typedef std::deque<iterator> heap_type
typedef heap_type::iterator heap_iterator
typedef adapt<UpdatePolicy, iterator> adapted_update_policy_type
typedef statistics_type::update_on_exit update_on_exit
```

**Private Members**

```cpp
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

template<>
adapt(Func f)

template<>
bool operator() (Iterator const &lhs, Iterator const &rhs) const

Public Members

template<>
Func f_

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 oper-
ation 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

typedef Key key_type

typedef Entry entry_type

typedef Statistics statistics_type

typedef std::pair<key_type, entry_type> entry_pair

typedef std::list<entry_pair> storage_type

typedef std::map<Key, typename storage_type::iterator> map_type

typedef std::size_t size_type
Public Functions

**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 &&other)*

**size_type size () const**

Return current size of the cache.

**Return** The current size of this cache instance.

**size_type capacity () const**

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*.

**Return** 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.

**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.

**bool holds_key** *(key_type const &key)*

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.

**Return** This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

**Parameters**

- **k**: [in] The key for the entry which should be looked up in the cache.

**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.

**Return** This function returns *true* if the cache holds the referenced entry, otherwise it returns *false*.

**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.
bool get_entry (key_type const &key, 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.
Return This function returns true if the cache holds the referenced entry, otherwise it returns
false.
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.

bool insert (key_type const &key, entry_type const &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.

void insert_nonexist (key_type const &key, entry_type const &entry)

void update (key_type const &key, entry_type const &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>
bool update_if (key_type const &key, entry_type const &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.
Return 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.
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.
template<typename Func>
size_type erase(Func const &ep)

## Return
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).

## 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`.

size_type erase()

## Return
Remove all stored entries from the cache.

size_type clear()

## Return
Clear the cache.

Unconditionally removes all stored entries from the cache.

statistics_type const &get_statistics() const

## Return
This function returns a reference to the statistics instance embedded inside this cache

## Private Types

typedef statistics_type::update_on_exit update_on_exit

## Private Functions

void touch(typename storage_type::iterator it)

void evict()
Private Members

size_type max_size_
size_type current_size_
storage_type storage_
map_type map_
statistics_type statistics_

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

typedef Value value_type

Public Functions

drop()  
Any cache entry has to be default constructible.

drop(value_type const &val)  
Construct a new instance of a cache entry holding the given value.

bool touch()  
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

Return 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.
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
    **Return** This function should return true if the entry should be added to the cache, otherwise
    it should return false.

bool remove()
    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
    **Return** 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, otherwise it will stay in the cache.

std::size_t get_size() const
    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.

value_type &get()
    Get a reference to the stored data value.

    **Note** This function is part of the CacheEntry concept

value_type const &get() const

**Private Members**

value_type value_

**Friends**

bool operator< (entry const &lhs, entry const &rhs)
    Forwarding operator< allowing to compare entries instead of the values.

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()**
Any cache entry has to be default constructible.

**fifo_entry(Value const &val)**
Construct a new instance of a cache entry holding the given value.

**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
**Return** This function should return true if the entry should be added to the cache, otherwise it should return false.

```cpp
std::chrono::steady_clock::time_point const &get_creation_time() const
```

### Private Types

```cpp
typedef entry<Value, fifo_entry<Value>> base_type
```

### Private Members

```cpp
std::chrono::steady_clock::time_point insertion_time_
```

### Friends

```cpp
bool operator<(fifo_entry const &lhs, fifo_entry const &rhs)
```

Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been created earlier (FIFO).

```cpp
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 ()
Any cache entry has to be default constructible.

lfu_entry (Value const &val)
Construct a new instance of a cache entry holding the given value.

bool touch ()
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<().

Return  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

unsigned long const &get_access_count () const

Private Types

typedef entry<Value, lfu_entry<Value>> base_type

Private Members

unsigned long ref_count_
### Friends

```cpp
default operator< (lfu_entry const &lhs, lfu_entry const &rhs)
    Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been accessed less frequently (LFU).
```

```cpp
namespace hpx
```

```cpp
namespace util
```

```cpp
namespace cache
```

```cpp
namespace entries
```

```cpp
template<typename Value>

class lru_entry : public hpx::util::cache::entries::entry<Value, lru_entry<Value> >
```

```cpp
#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 cache’s 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**

- `lru_entry()`
    Any cache entry has to be default constructible.

- `lru_entry (Value const &val)`
    Construct a new instance of a cache entry holding the given value.

- `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<()`.

**Return** 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.

- `std::chrono::steady_clock::time_point const &get_access_time () const`
    Returns the last access time of the entry.
Private Types

typedef entry<Value, lru_entry<Value>> base_type

Private Members

\[std::chrono::steady_clock::time_point access_time_\]

Friends

bool operator< (lru_entry const &lhs, lru_entry const &rhs)
Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has been accessed less recently (LRU).

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 ()
Any cache entry has to be default constructible.

size_entry (Value const &val, std::size_t size)
Construct a new instance of a cache entry holding the given value.

std::size_t get_size () const
Return the ‘size’ of this entry.
Private Types

typedef detail::size_derived<Value, Derived>::type derived_type
typedef entry<Value, derived_type> base_type

Private Members

std::size_t size_

Friends

bool operator< (size_entry const &lhs, size_entry const &rhs)
   Compare the ‘age’ of two entries. An entry is ‘older’ than another entry if it has a bigger size.

namespace hpx

namespace util

namespace cache

namespace policies

template<typename Entry>
struct always

Public Functions

bool operator() (Entry const&)

namespace hpx

namespace util

namespace cache

namespace statistics

class local_full_statistics : public hpx::util::cache::statistics::local_statistics
Public Functions

\texttt{std::int64_t get\_get\_entry\_count (bool reset)}

The function \texttt{get\_get\_entry\_count} returns the number of invocations of the \texttt{get\_entry()} API function of the cache.

\texttt{std::int64_t get\_insert\_entry\_count (bool reset)}

The function \texttt{get\_insert\_entry\_count} returns the number of invocations of the \texttt{insert\_entry()} API function of the cache.

\texttt{std::int64_t get\_update\_entry\_count (bool reset)}

The function \texttt{get\_update\_entry\_count} returns the number of invocations of the \texttt{update\_entry()} API function of the cache.

\texttt{std::int64_t get\_erase\_entry\_count (bool reset)}

The function \texttt{get\_erase\_entry\_count} returns the number of invocations of the \texttt{erase()} API function of the cache.

\texttt{std::int64_t get\_get\_entry\_time (bool reset)}

The function \texttt{get\_get\_entry\_time} returns the overall time spent executing of the \texttt{get\_entry()} API function of the cache.

\texttt{std::int64_t get\_insert\_entry\_time (bool reset)}

The function \texttt{get\_insert\_entry\_time} returns the overall time spent executing of the \texttt{insert\_entry()} API function of the cache.

\texttt{std::int64_t get\_update\_entry\_time (bool reset)}

The function \texttt{get\_update\_entry\_time} returns the overall time spent executing of the \texttt{update\_entry()} API function of the cache.

\texttt{std::int64_t get\_erase\_entry\_time (bool reset)}

The function \texttt{get\_erase\_entry\_time} returns the overall time spent executing of the \texttt{erase()} API function of the cache.

Private Functions

\texttt{std::int64_t get\_and\_reset\_value (std::int64_t \&value, bool reset)}

Private Members

\texttt{api\_counter\_data get\_entry\_\_} 

\texttt{api\_counter\_data insert\_entry\_\_} 

\texttt{api\_counter\_data update\_entry\_} 

\texttt{api\_counter\_data erase\_entry\_}
struct api_counter_data

Public Functions

api_counter_data()

Public Members

std::int64_t count_
std::int64_t time_

struct update_on_exit
#include <local_full_statistics.hpp> Helper class to update timings and counts on function exit.

Public Functions

update_on_exit (local_full_statistics &stat, method m)
~update_on_exit()

Public Members

std::int64_t started_at_
api_counter_data &data_

Private Static Functions

static api_counter_data &get_api_counter_data (local_full_statistics &stat, method m)

namespace hpx

namespace util

namespace cache

namespace statistics

class local_statistics : public hpx::util::cache::statistics::no_statistics
    Subclassed by hpx::util::cache::statistics::local_full_statistics
Public Functions

\texttt{local\_statistics()}  
\textit{std::size\_t get\_and\_reset (std::size\_t \&value, bool reset)}  
\textit{std::size\_t hits (const)}  
\textit{std::size\_t misses (const)}  
\textit{std::size\_t insertions (const)}  
\textit{std::size\_t evictions (const)}  
\textit{std::size\_t hits (bool reset)}  
\textit{std::size\_t misses (bool reset)}  
\textit{std::size\_t insertions (bool reset)}  
\textit{std::size\_t evictions (bool reset)}  

\textit{void got\_hit ()}  
\hspace{1em} The function \textit{got\_hit} will be called by a cache instance whenever a entry got touched.

\textit{void got\_miss ()}  
\hspace{1em} The function \textit{got\_miss} will be called by a cache instance whenever a requested entry has not been found in the cache.

\textit{void got\_insertion ()}  
\hspace{1em} The function \textit{got\_insertion} will be called by a cache instance whenever a new entry has been inserted.

\textit{void got\_eviction ()}  
\hspace{1em} The function \textit{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.

\textit{void clear ()}  
\hspace{1em} Reset all statistics.

Private Members

\textit{std::size\_t hits_}  
\textit{std::size\_t misses_}  
\textit{std::size\_t insertions_}  
\textit{std::size\_t evictions_}  

\texttt{namespace hpx}  
\texttt{namespace util}  
\texttt{namespace cache}  
\texttt{namespace statistics}
Enums

enum method

Values:

method_get_entry = 0
method_insert_entry = 1
method_update_entry = 2
method_erase_entry = 3

class no_statistics

Subclassed by hpx::util::cache::statistics::local_statistics

Public Functions

void got_hit()

The function got_hit will be called by a cache instance whenever a entry got touched.

void got_miss()

The function got_miss will be called by a cache instance whenever a requested entry has not been found in the cache.

void got_insertion()

The function got_insertion will be called by a cache instance whenever a new entry has been inserted.

void got_eviction()

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.

void clear()

Reset all statistics.

int64_t get_get_entry_count(bool)

The function get_get_entry_count returns the number of invocations of the get_entry() API function of the cache.

int64_t get_insert_entry_count(bool)

The function get_insert_entry_count returns the number of invocations of the insert_entry() API function of the cache.

int64_t get_update_entry_count(bool)

The function get_update_entry_count returns the number of invocations of the update_entry() API function of the cache.

int64_t get_erase_entry_count(bool)

The function get_erase_entry_count returns the number of invocations of the erase() API function of the cache.

int64_t get_get_entry_time(bool)

The function get_get_entry_time returns the overall time spent executing of the get_entry() API function of the cache.
The function `get_insert_entry_time` returns the overall time spent executing of the `insert_entry()` API function of the cache.

The function `get_update_entry_time` returns the overall time spent executing of the `update_entry()` API function of the cache.

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

**update_on_exit** *(no_statistics const&, method)*

---

**command_line_handling_local**

The contents of this module can be included with the header `hpx/modules/command_line_handling_local.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/command_line_handling_local.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace util

 Enums

    enum commandline_error_mode
    Values:
        return_on_error
        rethrow_on_error
        allow_unregistered
        ignore_aliases = 0x40
        report_missing_config_file = 0x80
```
concepts

The contents of this module can be included with the header `hpx/modules/concepts.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/concepts.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

```cpp
HPX_CONCEPT_REQUIRES_(...) 
HPX_CONCEPT_REQUIRES(...) 
HPX_CONCEPT_ASSERT(...) 
```

Defines

```cpp
HPX_HAS_MEMBER_XXX_TRAIT_DEF(MEMBER) 
This macro creates a boolean unary meta-function which result is true if and only if its parameter type has member function with MEMBER name (no matter static it is or not). The generated trait ends up in a namespace where the macro itself has been placed.
```

Defines

```cpp
HPX_HAS_XXX_TRAIT_DEF(Name) 
This macro creates a boolean unary meta-function such that for any type X, has_name<X>::value == true if and only if X is a class type and has a nested type member x::name. The generated trait ends up in a namespace where the macro itself has been placed.
```

concurrency

The contents of this module can be included with the header `hpx/modules/concurrency.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/concurrency.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx {
    namespace util {
        class barrier {
        }
    }
}
```
Public Functions

\texttt{barrier (std::size\_t number\_of\_threads)}

\texttt{\sim barrier ()}

\texttt{void wait ()}

Private Types

\texttt{typedef std::mutex mutex\_type}

Private Members

\texttt{const std::size\_t number\_of\_threads}\_
\texttt{std::size\_t total}\_
\texttt{mutex\_type mtx\_}
\texttt{std::condition\_variable cond}\_

Private Static Attributes

\texttt{constexpr std::size\_t barrier\_flag = static\_cast<std::size\_t>(1) \ll (CHAR\_BIT * sizeof(std::size\_t) - 1)}

template<typename Data>
\texttt{struct cache\_aligned\_data\langle Data, std::false\_type \rangle}

Public Functions

\texttt{cache\_aligned\_data ()}

\texttt{cache\_aligned\_data (Data &&data)}

\texttt{cache\_aligned\_data (Data const &data)}

Public Members

Data data_

template<typename Data>
\texttt{struct cache\_aligned\_data\_derived\langle Data, std::false\_type \rangle : public Data}
Public Functions

```cpp
cache_aligned_data_derived()
cache_aligned_data_derived(Data &&data)
cache_aligned_data_derived(Data const &data)
```

global namespace hpx

global namespace threads

Functions

```cpp
constexpr std::size_t get_cache_line_size()
```

global namespace util

Typedefs

```cpp
template<typename Data>
using cache_line_data = cache_aligned_data<Data>

template<typename Data, typename NeedsPadding = typename detail::needs_padding<Data>::type>
struct cache_aligned_data
```

Public Functions

```cpp
cache_aligned_data()
cache_aligned_data(Data &&data)
cache_aligned_data(Data const &data)
```

Public Members

```cpp
Data data_

template<>
char cacheline_pad[get_cache_line_padding_size(sizeof(Data))]

template<typename Data>
struct cache_aligned_data<Data, std::false_type>
```
Public Functions

`cache_aligned_data()`

`cache_aligned_data`(Data &&data)

`cache_aligned_data`(Data const &data)

Public Members

Data `data`

template<typename Data, typename NeedsPadding = typename detail::needs_padding<Data>::type>
struct cache_aligned_data_derived : public Data

Public Functions

`cache_aligned_data_derived()`

`cache_aligned_data_derived`(Data &&data)

`cache_aligned_data_derived`(Data const &data)

Public Members

template<>
char cacheline_pad[get_cache_line_padding_size(sizeof(Data))]

template<typename Data>
struct cache_aligned_data_derived<Data, std::false_type> : public Data

Public Functions

`cache_aligned_data_derived()`

`cache_aligned_data_derived`(Data &&data)

`cache_aligned_data_derived`(Data const &data)

Defines

MOODYCAMEL_THREADLOCAL

MOODYCAMEL_EXCEPTIONS_ENABLED

MOODYCAMEL_TRY

MOODYCAMEL_CATCH(...) 

MOODYCAMEL_RETHROW

MOODYCAMEL_THROW(expr)

MOODYCAMEL_NOEXCEPT
namespace hpx

namespace concurrency

Functions

template<typename T, typename Traits>
void swap (typename ConcurrentQueue< T, Traits >::ImplicitProducerKVP & a, typename ConcurrentQueue< T, Traits >::ImplicitProducerKVP & b)

template<typename T, typename Traits>
void swap (ConcurrentQueue< T, Traits > & a, ConcurrentQueue< T, Traits > & b)

void swap (ProducerToken & a, ProducerToken & b)

void swap (ConsumerToken & a, ConsumerToken & b)

template<typename T, typename Traits = ConcurrentQueueDefaultTraits>
class ConcurrentQueue

Public Types

typedef ::hpx::concurrency::ProducerToken producer_token_t

typedef ::hpx::concurrency::ConsumerToken consumer_token_t

typedef Traits::index_t index_t

typedef Traits::size_t size_t

Public Functions

ConcurrentQueue (size_t capacity = 6 * BLOCK_SIZE)

ConcurrentQueue (size_t minCapacity, size_t maxExplicitProducers, size_t maxImplicitProducers)

~ConcurrentQueue ()

ConcurrentQueue (ConcurrentQueue const&)

ConcurrentQueue & operator= (ConcurrentQueue const&)

ConcurrentQueue (ConcurrentQueue & other)

ConcurrentQueue & operator= (ConcurrentQueue & other)

void swap (ConcurrentQueue & other)

bool enqueue (T const & item)
bool enqueue (T &&item)

bool enqueue (producer_token_t const &token, T const &item)

bool enqueue (producer_token_t const &token, T &&item)

template< typename It>
bool enqueue_bulk (It itemFirst, size_t count)

template< typename It>
bool enqueue_bulk (producer_token_t const &token, It itemFirst, size_t count)

bool try_enqueue (T const &item)

bool try_enqueue (T &&item)

bool try_enqueue (producer_token_t const &token, T const &item)

bool try_enqueue (producer_token_t const &token, T &&item)

template< typename It>
bool try_enqueue_bulk (It itemFirst, size_t count)

template< typename It>
bool try_enqueue_bulk (producer_token_t const &token, It itemFirst, size_t count)

template< typename U>
bool try_dequeue (U &item)

bool try_dequeue_non_interleaved (U &item)

bool try_dequeue (consumer_token_t &token, U &item)

template< typename It>
size_t try_dequeue_bulk (It itemFirst, size_t max)

template< typename It>
size_t try_dequeue_bulk (consumer_token_t &token, It itemFirst, size_t max)

template< typename U>
bool try_dequeue_from_producer (producer_token_t const &producer, U &item)

template< typename It>
size_t try_dequeue_bulk_from_producer (producer_token_t const &producer, It itemFirst, size_t max)

size_t size_approx () const
Public Static Functions

static bool is_lock_free()

Public Static Attributes

const size_t BLOCK_SIZE = static_cast<size_t>(Traits::BLOCK_SIZE)
const size_t EXPLICIT_BLOCK_EMPTY_COUNTER_THRESHOLD = static_cast<size_t>(Traits::EXPLICIT_BLOCK_EMPTY_COUNTER_THRESHOLD)
const size_t EXPLICITInicial_INDEX_SIZE = static_cast<size_t>(Traits::EXPLICIT_INITIAL_INDEX_SIZE)
const size_t IMPLICIT_INITIAL_INDEX_SIZE = static_cast<size_t>(Traits::IMPLICIT_INITIAL_INDEX_SIZE)
const size_t INITIAL_IMPLICIT_PRODUCER_HASH_SIZE = static_cast<size_t>(Traits::INITIAL_IMPLICIT_PRODUCER_HASH_SIZE)
const std::uint32_t EXPLICIT_CONSUMER_CONSUMPTION_QUOTA_BEFORE_ROTATE = static_cast<std::uint32_t>(Traits::EXPLICIT_CONSUMER_CONSUMPTION_QUOTA_BEFORE_ROTATE)

Private Types

enum AllocationMode

Values:

CanAlloc
CannotAlloc

enum InnerQueueContext

Values:

implicit_context = 0
explicit_context = 1

Private Functions

ConcurrentQueue & swap_internal (ConcurrentQueue & other)

template<AllocationMode canAlloc, typename U>
bool inner_enqueue (producer_token_t const &token, U &&element)

template<AllocationMode canAlloc, typename U>
bool inner_enqueue (U &&element)

template<AllocationMode canAlloc, typename It>
bool inner_enqueue_bulk (producer_token_t const &token, It itemFirst, size_t count)

template<AllocationMode canAlloc, typename It>
bool inner_enqueue_bulk (It itemFirst, size_t count)

bool update_current_producer_after_rotation (consumer_token_t &token)

void populate_initial_block_list (size_t blockCount)

Block * try_get_block_from_initial_pool ()

void add_block_to_free_list (Block *block)
void add_blocks_to_free_list (Block *block)

Block *try_get_block_from_free_list()

template<AllocationMode canAlloc>
Block *requisition_block()

ProducerBase *recycle_or_create_producer (bool isExplicit)

ProducerBase *recycle_or_create_producer (bool isExplicit, bool &recycled)

ProducerBase *add_producer (ProducerBase *producer)

void reown_producers()

void populate_initial_implicit_producer_hash()

void swap_implicitProducer_hashes (ConcurrentQueue &other)

ImplicitProducer *get_or_add_implicitProducer()

Private Members

std::atomic<ProducerBase*> producerListTail
std::atomic<std::uint32_t> producerCount
std::atomic<size_t> initialBlockPoolIndex
Block *initialBlockPool
size_t initialBlockPoolSize
FreeList<Block> freeList
std::atomic<ImplicitProducerHash*> implicitProducerHash
std::atomic<size_t> implicitProducerHashCount
ImplicitProducerHash initialImplicitProducerHash
std::array<ImplicitProducerKVP, INITIAL_IMPLICIT_PRODUCER_HASH_SIZE> initialImplicitProducerHashEntries
std::atomic_flag implicitProducerHashResizeInProgress
std::atomic<std::uint32_t> nextExplicitConsumerId
std::atomic<std::uint32_t> globalExplicitConsumerOffset

Private Static Functions

template< typename U>
static U *create_array (size_t count)

template< typename U>
static void destroy_array (U *p, size_t count)

template< typename U>
static U *create ()

template< typename U, typename A1>
static $U*create(A1&&a1)$

template<typename $U$>
static void destroy ($U*p$)

**Friends**

friend hpx::concurrency::ProducerToken
friend hpx::concurrency::ConsumerToken
friend hpx::concurrency::ExplicitProducer
friend hpx::concurrency::ImplicitProducer
friend hpx::concurrency::ConcurrentQueueTests

template<typename $XT$, typename $XTraits$>
void swap (typename ConcurrentQueue<$XT$, $XTraits$>::ImplicitProducerKVP&, typename ConcurrentQueue<$XT$, $XTTraits$>::ImplicitProducerKVP&)

struct Block

**Public Functions**

template<>
Block()

template<InnerQueueContext $context$>
bool is_empty () const

template<InnerQueueContext $context$>
bool set_empty (index_t $i$)

template<InnerQueueContext $context$>
bool set_many_empty (index_t $i$, size_t $count$)

template<InnerQueueContext $context$>
void set_all_empty ()

template<InnerQueueContext $context$>
void reset_empty ()

template<>
T *operator[] (index_t $idx$)

template<>
T const *operator[] (index_t $idx$) const
Public Members

template<>
char elements[sizeof(T) * BLOCK_SIZE]

template<>
details::max_align_t dummy

template<>
Block *next

template<>
std::atomic<size_t> elementsCompletelyDequeued

std::atomic<bool> hpx::concurrency::ConcurrentQueue<T, Traits>::Block::emptyFlags[
BLOCK_SIZE<=EXPLICIT_BLOCK_EMPTY_COUNTER_THRESHOLD ? BLOCK_SIZE :1]

template<>
std::atomic<std::uint32_t> freeListRefs

template<>
std::atomic<Block*> freeListNext

template<>
std::atomic<bool> shouldBeOnFreeList

template<>
bool dynamicallyAllocated

Private Members

template<>
union hpx::concurrency::ConcurrentQueue::Block::[anonymous] [anonymous]

struct ExplicitProducer : public hpx::concurrency::ConcurrentQueue<T, Traits>::ProducerBase

Public Functions

template<>
ExplicitProducer(ConcurrentQueue *parent)

template<>
~ExplicitProducer()

template<AllocationMode allocMode, typename U>
bool enqueue(U &&element)

template<typename U>
bool dequeue(U &element)

template<AllocationMode allocMode, typename It>
bool enqueue_bulk(It itemFirst, size_t count)

template<typename It>
size_t dequeue_bulk(It &itemFirst, size_t max)
Private Functions

template<>
bool new_block_index(size_t numberOfFilledSlotsToExpose)

Private Members

template<>
std::atomic<BlockIndexHeader*> blockIndex

template<>
size_t pr_blockIndexSlotsUsed

template<>
size_t pr_blockIndexSize

template<>
size_t pr_blockIndexFront

template<>
BlockIndexEntry *pr_blockIndexEntries

template<>
void *pr_blockIndexRaw

struct BlockIndexEntry

Public Members

template<>
index_t base

template<>
Block *block

struct BlockIndexHeader

Public Members

template<>
size_t size

template<>
std::atomic<size_t> front

template<>
BlockIndexEntry *entries

template<>
void *prev

template<typename N>
struct FreeList
Public Functions

```cpp
template<> FreeList ()

template<> FreeList (FreeList &&other)

template<> void swap (FreeList &other)

template<> FreeList (FreeList const&)

template<> FreeList &operator= (FreeList const&)

template<> void add (N *node)

template<> N *try_get ()

template<> N *head_unsafe () const
```

Private Functions

```cpp
template<> void add_knowing_refcount_is_zero (N *node)
```

Private Members

```cpp
template<>
std::atomic<N*> freeListHead
```

Private Static Attributes

```cpp
template<>
const std::uint32_t REFS_MASK = 0x7FFFFFFF

template<>
const std::uint32_t SHOULD_BE_ON_FREELIST = 0x80000000
```

template<typename N>
struct FreeListNode
Public Functions

template<>
FreeListNode();

Public Members

template<>
std::atomic<std::uint32_t> freeListRefs

template<>
std::atomic<N*> freeListNext

struct ImplicitProducer : public hpx::concurrency::ConcurrentQueue<T, Traits>::ProducerBase

Public Functions

template<>
ImplicitProducer (ConcurrentQueue *parent)

template<>
~ImplicitProducer();

template<AllocationMode allocMode, typename U>
bool enqueue (U &element)

template<typename U>
bool dequeue (U &element)

template<AllocationMode allocMode, typename It>
bool enqueue_bulk (It itemFirst, size_t count)

template<typename It>
size_t dequeue_bulk (It &itemFirst, size_t max)

Private Functions

template<AllocationMode allocMode>
bool insert_block_index_entry (BlockIndexEntry *&idxEntry, index_t blockStartIndex)

template<>
void rewind_block_index_tail();

template<>
BlockIndexEntry *get_block_index_entry_for_index (index_t index) const

template<>
size_t get_block_index_index_for_index (index_t index, BlockIndexHeader *localBlockIndex) const

template<>
bool new_block_index();
Private Members

template<>
size_t nextBlockIndexCapacity

template<>
std::atomic<BlockIndexHeader*> blockIndex

Private Static Attributes

template<>
const index_t INVALID_BLOCK_BASE = 1

struct BlockIndexEntry

Public Members

template<>
std::atomic<index_t> key

template<>
std::atomic<Block*> value

struct BlockIndexHeader

Public Members

template<>
size_t capacity

template<>
std::atomic<size_t> tail

template<>
BlockIndexEntry* entries

template<>
BlockIndexEntry** index

template<>
BlockIndexHeader* prev

struct ImplicitProducerHash

Public Members

template<>
size_t capacity

template<>
ImplicitProducerKVP* entries

template<>
ImplicitProducerHash* prev

struct ImplicitProducerKVP
Public Functions

template<>
ImplicitProducerKVP() 

template<>
ImplicitProducerKVP(ImplicitProducerKVP &&other) 

template<>
ImplicitProducerKVP &operator=(ImplicitProducerKVP &&other) 

template<>
void swap(ImplicitProducerKVP &other) 

Public Members

template<>
std::atomic<details::thread_id_t> key 

template<>
ImplicitProducer *value 

struct ProducerBase : public hpx::concurrency::details::ConcurrentQueueProducerTypelessBase 

Public Functions

template<>
ProducerBase(ConcurrentQueue *parent_, bool isExplicit_) 

template<>
virtual ~ProducerBase() 

template<typename U>
bool dequeue(U &element) 

template<typename It>
size_t dequeue_bulk(It &itemFirst, size_t max) 

template<>
ProducerBase *next_prod() const 

template<>
size_t size_approx() const 

template<>
index_t getTail() const
Public Members

template<>
bool isExplicit

template<>
ConcurrentQueue *parent

Protected Attributes

template<>
std::atomic<index_t> tailIndex

template<>
std::atomic<index_t> headIndex

template<>
std::atomic<index_t> dequeueOptimisticCount

template<>
std::atomic<index_t> dequeueOvercommit

template<>
Block *tailBlock

struct ConcurrentQueueDefaultTraits

Public Types

typedef std::size_t size_t

typedef std::size_t index_t

Public Static Functions

static void *malloc (size_t size)

static void free (void *ptr)

Public Static Attributes

const size_t BLOCK_SIZE = 32
const size_t EXPLICIT_BLOCK_EMPTY_COUNTER_THRESHOLD = 32
const size_t EXPLICIT_INITIAL_INDEX_SIZE = 32
const size_t IMPLICIT_INITIAL_INDEX_SIZE = 32
const size_t INITIAL_IMPLICIT_PRODUCER_HASH_SIZE = 32
const std::uint32_t EXPLICIT_CONSUMER_CONSUMPTION_QUOTA_BEFORE_ROTATE = 256
const size_t MAX_SUBQUEUE_SIZE = details::const_numeric_max<size_t>::value

struct ConsumerToken
**Public Functions**

```cpp
template<typename T, typename Traits>
ConsumerToken (ConcurrentQueue<T, Traits> &q)
```

```cpp
template<typename T, typename Traits>
ConsumerToken (BlockingConcurrentQueue<T, Traits> &q)
```

```cpp
ConsumerToken (ConsumerToken &&other)
```

```cpp
ConsumerToken &operator= (ConsumerToken &&other)
```

```cpp
void swap (ConsumerToken &other)
```

```cpp
ConsumerToken (ConsumerToken const&)
```

```cpp
ConsumerToken &operator= (ConsumerToken const&)
```

**Private Members**

```cpp
std::uint32_t initialOffset
```

```cpp
std::uint32_t lastKnownGlobalOffset
```

```cpp
std::uint32_t itemsConsumedFromCurrent
```

```cpp
details::ConcurrentQueueProducerTypelessBase *currentProducer
```

```cpp
details::ConcurrentQueueProducerTypelessBase *desiredProducer
```

**Friends**

```cpp
friend hpx::concurrency::ConcurrentQueue
```

```cpp
friend hpx::concurrency::ConcurrentQueueTests
```

```cpp
struct ProducerToken
```

**Public Functions**

```cpp
template<typename T, typename Traits>
ProducerToken (ConcurrentQueue<T, Traits> &queue)
```

```cpp
template<typename T, typename Traits>
ProducerToken (BlockingConcurrentQueue<T, Traits> &queue)
```

```cpp
ProducerToken (ProducerToken &&other)
```

```cpp
ProducerToken &operator= (ProducerToken &&other)
```

```cpp
void swap (ProducerToken &other)
```

```cpp
bool valid () const
```

```cpp
~ProducerToken ()
```

```cpp
ProducerToken (ProducerToken const&)
```

```cpp
ProducerToken &operator= (ProducerToken const&)
```
Protected Attributes

`details::ConcurrentQueueProducerTypelessBase *producer`

Friends

`friend hpx::concurrency::ConcurrentQueue`  
`friend hpx::concurrency::ConcurrentQueueTests`

namespace details

Typedefs

`typedef std::uintptr_t thread_id_t`  
`typedef std::max_align_t std_max_align_t`

Functions

`static thread_id_t thread_id()`  
`static bool() hpx::concurrency::details::likely(bool x)`  
`static bool() hpx::concurrency::details::unlikely(bool x)`  
`static size_t hash_thread_id(thread_id_t id)`  
`template<typename T> static bool circular_less_than(T a, T b)`  
`template<typename U> static char *align_for(char *ptr)`  
`template<typename T> static T ceil_to_pow_2(T x)`  
`template<typename T> static void swap_relaxed(std::atomic<T> &left, std::atomic<T> &right)`  
`template<typename T> static T const &nomove(T const &x)`  
`template<typename It> static auto deref_noexcept(It &it)`
Variables

```cpp
const thread_id_t invalid_thread_id = 0
const thread_id_t invalid_thread_id2 = 1
```

template<bool use32>
```cpp
struct _hash_32_or_64
```

Public Static Functions

```cpp
static std::uint32_t hash (std::uint32_t h)
```

template<>
```cpp
struct _hash_32_or_64<1>
```

Public Static Functions

```cpp
static std::uint64_t hash (std::uint64_t h)
```

```
struct ConcurrentQueueProducerTypelessBase
```

Public Functions

```cpp
ConcurrentQueueProducerTypelessBase()
```

Public Members

```cpp
ConcurrentQueueProducerTypelessBase *next
std::atomic<bool> inactive
ProducerToken *token
```

template<typename T>
```cpp
struct const_numeric_max
```

Public Static Attributes

```cpp
const T hpx::concurrency::details::const_numeric_max::value= std::numeric_limits<T>::is_signed ? (static_cast<T>(1) << (sizeof(T) * CHAR_BIT - 1)) - static_cast<T>(1) : static_cast<T>(-1)
```

union max_align_t
Public Members

`std_max_align_t x`
long long `y`
void *`z`

template<bool Enable>
`struct nomove_if`

Public Static Functions

template<typename T>
`static T const &eval(T const &x)`

template<>
`struct nomove_if<false>`

Public Static Functions

template<typename U>
`static auto eval(U &&x)`

template<>
`struct static_is_lock_free<bool>`

Public Types

```cpp
enum [anonymous]
Values:
value = ATOMIC_BOOL_LOCK_FREE
```

template<typename U>
`struct static_is_lock_free<U*>`

Public Types

```cpp
enum [anonymous]
Values:
value = ATOMIC_POINTER_LOCK_FREE
```

template<typename T>
`struct static_is_lock_free_num`
Public Types

enum [anonymous]
Values:

value = 0

template<> 
struct static_is_lock_free_num<int>

Public Types

enum [anonymous]
Values:

value = ATOMIC_INT_LOCK_FREE

template<> 
struct static_is_lock_free_num<long>

Public Types

enum [anonymous]
Values:

value = ATOMIC_LONG_LOCK_FREE

template<> 
struct static_is_lock_free_num<long long>

Public Types

enum [anonymous]
Values:

value = ATOMIC_LLONG_LOCK_FREE

template<> 
struct static_is_lock_free_num<short>

Public Types

enum [anonymous]
Values:

value = ATOMIC_SHORT_LOCK_FREE

template<> 
struct static_is_lock_free_num<signed char>
Public Types

enum [anonymous]
Values:
value = ATOMIC_CHAR_LOCK_FREE

template<typename thread_id_t>
struct thread_id_converter

Public Types

typedef thread_id_t thread_id_numeric_size_t
typedef thread_id_t thread_id_hash_t

Public Static Functions

static thread_id_hash_t prehash (thread_id_t const &x)

namespace boost

namespace lockfree

Enums

enum deque_status_type
Values:
stable
rpush
lpush

template<typename T, typename freelist_t = caching_freelist_t, typename Alloc = std::allocator<T>>
struct deque

Public Types

template<>
using node = deque_node<T>
template<>
using node_pointer = typename node::pointer
template<>
using atomic_node_pointer = typename node::atomic_pointer
template<>
using tag_t = typename node::tag_t
template<>
using anchor = deque_anchor<T>
template<>
using anchor_pair = typename anchor::pair
template<>
using atomic_anchor_pair = typename anchor::atomic_pair
template<>
using node_allocator = typename std::allocator_traits<Alloc>::template rebind_alloc<node>
template<>
using pool = typename std::conditional<std::is_same<freelist_t, caching_freelist_t>::value, caching_freelist<node>, static_freelist<node> >::type

Public Functions

HPX_NON_COPYABLE (deque)
deque (std::size_t initial_nodes = 128)
~deque ()
bool empty () const
bool is_lock_free () const
bool push_left (T data)
bool push_right (T data)
bool pop_left (T &r)
bool pop_left (T *r)
bool pop_right (T &r)
bool pop_right (T *r)

Private Functions

node *alloc_node (node *lptr, node *rptr, T const &v, tag_t ltag = 0, tag_t rtag = 0)
node *alloc_node (node *lptr, node *rptr, T &&v, tag_t ltag = 0, tag_t rtag = 0)
void dealloc_node (node *n)
void stabilize_left (anchor_pair &lrs)
void stabilize_right (anchor_pair &lrs)
void stabilize (anchor_pair &lrs)
**Private Members**

anchor anchor_
pool pool_
template<> char padding[padding_size]

**Private Static Attributes**

\[
\text{constexpr std::size_t padding_size} = \text{BOOST\_LOCKFREE\_CACHELINE\_BYTES} - \text{sizeof(anchor)}
\]

template<typename T>
struct deque_anchor

**Public Types**

template<>
using node = deque_node<T>
template<>
using node_pointer = typename node::pointer
template<>
using atomic_node_pointer = typename node::atomic_pointer
template<>
using tag_t = typename node::tag_t
template<>
using anchor = deque_anchor<T>
template<>
using pair = tagged_ptr_pair<node, node>
template<>
using atomic_pair = std::atomic<pair>

**Public Functions**

deque_anchor ()
deque_anchor (deque_anchor const &p)
deque_anchor (pair const &p)
deque_anchor (node *lptr, node *rptr, tag_t status = stable, tag_t tag = 0)
pair lrs (std::memory_order mo = std::memory_order_acquire) volatile const
deque <std::memory_order mo = std::memory_order_acquire) volatile const
deque <std::memory_order mo = std::memory_order_acquire) volatile const
deque <std::memory_order mo = std::memory_order_acquire) volatile const
bool cas (deque_anchor &expected, deque_anchor const &desired, std::memory_order mo = std::memory_order_acq_rel) volatile
bool cas (pair &expected, deque_anchor const &desired, std::memory_order mo = std::memory_order_acq_rel) volatile
bool cas (deque_anchor &expected, pair const &desired, std::memory_order mo = std::memory_order_acq_rel) volatile
bool cas (pair &expected, pair const &desired, std::memory_order mo = std::memory_order_acq_rel) volatile
bool operator==(volatile deque_anchor const &rhs) const
bool operator!=(volatile deque_anchor const &rhs) const
bool operator==(volatile pair const &rhs) const
bool operator!=(volatile pair const &rhs) const
bool is_lock_free() const

Private Members

atomic_pair pair_

template<typename T>
struct deque_node

Public Types

typedef detail::tagged_ptr<deque_node> pointer
typedef std::atomic<pointer> atomic_pointer
typedef pointer::tag_t tag_t

Public Functions

deleque_node ()
deleque_node (deque_node const &p)
deleque_node (deque_node *lptr, deque_node *rptr, T const &v, tag_t ltag = 0, tag_t rtag = 0)
deleque_node (deque_node *lptr, deque_node *rptr, T &&v, tag_t ltag = 0, tag_t rtag = 0)
Public Members

atomic_pointer left
atomic_pointer right
T data

namespace hpx

namespace util

struct spinlock
#include <spinlock.hpp> Lockable spinlock class.

Public Functions

HPX_NON_COPYABLE (spinlock)
spinlock (char const* = nullptr)
~spinlock ()
void lock ()
bool try_lock ()
void unlock ()

Private Members

hpx::util::detail::spinlock m

namespace hpx

namespace util

template<typename Tag, std::size_t N = HPX_HAVE_SPINLOCK_POOL_NUM>
class spinlock_pool

Public Static Functions

static detail::spinlock &spinlock_for (void const *pv)
Private Static Attributes

```
cache_aligned_data<detail::spinlock> pool_
```

**config**

The contents of this module can be included with the header `hpx/modules/config.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/config.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

**Defines**

**HPX_INITIAL_IP_PORT**
This is the default ip/port number used by the parcel subsystem.

**HPX_CONNECTING_IP_PORT**

**HPX_INITIAL_IP_ADDRESS**

**HPX_RUNTIME_INSTANCE_LIMIT**
This defines the maximum number of possible runtime instances in one executable.

**HPX_PARCEL_BOOTSTRAP**
This defines the type of the parcelport to be used during application bootstrap. This value can be changed at runtime by the configuration parameter:

```
hpx.parcel.bootstrap = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_BOOTSTRAP).

**HPX_PARCEL_MAX_CONNECTIONS**
This defines the number of outgoing (parcel-) connections kept alive (to all other localities). This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_connections = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_MAX_CONNECTIONS).

**HPX_PARCEL_IPC_DATA_BUFFER_CACHE_SIZE**
This defines the number of outgoing ipc (parcel-) connections kept alive (to each of the other localities on the same node). This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.ipc.data_buffer_cache_size = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_IPC_DATA_BUFFER_CACHE_SIZE).

**HPX_PARCEL_MPI_MAX_REQUESTS**
This defines the number of MPI requests in flight. This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.mpi.max_requests = ...
```

(or by setting the corresponding environment variable HPX_PARCEL_MPI_MAX_REQUESTS).

**HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY**
This defines the number of outgoing (parcel-) connections kept alive (to each of the other localities). This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_connections_per_locality = ...
```
HPX Documentation, master

(or by setting the corresponding environment variable HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY).

**HPX_PARCEL_MAX_MESSAGE_SIZE**
This defines the maximally allowed message size for messages transferred between localities. This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_message_size = ...```
(or by setting the corresponding environment variable HPX_PARCEL_MAX_MESSAGE_SIZE).

**HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE**
This defines the maximally allowed outbound message size for coalescing messages transferred between localities. This value can be changed at runtime by setting the configuration parameter:

```
hpx.parcel.max_outbound_message_size = ...```
(or by setting the corresponding environment variable HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE).

**HPX_PARCEL Serialization Overhead**

**HPX_AGAS_LOCAL_CACHE_SIZE**
This defines the number of AGAS address translations kept in the local cache. This is just the initial size which may be adjusted depending on the load of the system (not implemented yet), etc. It must be a minimum of 3 for AGAS v3 bootstrapping.

This value can be changed at runtime by setting the configuration parameter:

```
hpx.agas.local_cache_size = ...```
(or by setting the corresponding environment variable HPX_AGAS_LOCAL_CACHE_SIZE)

**HPX_INITIAL_AGAS_MAX_PENDING_REFCNT_REQUESTS**

**HPX_GLOBALCREDIT_INITIAL**
This defines the initial global reference count associated with any created object.

**HPX_NUM_IO_POOL_SIZE**
This defines the default number of OS-threads created for the different internal thread pools

**HPX_NUM_PARCEL_POOL_SIZE**

**HPX_NUM_TIMER_POOL_SIZE**

**HPX_SPINLOCK_DEADLOCK_DETECTION_LIMIT**
By default, enable minimal thread deadlock detection in debug builds only.

**HPX_COROUTINE_NUM_HEAPS**
This defines the default number of coroutine heaps.

**HPX_HAVE_THREAD_BACKTRACE_DEPTH**
By default, enable storing the thread phase in debug builds only.

By default, enable storing the parent thread information in debug builds only. By default, enable storing the thread description in debug builds only. By default, enable storing the target address of the data the thread is accessing in debug builds only. By default we do not maintain stack back-traces on suspension. This is a pure debugging aid to be able to see in the debugger where a suspended thread got stuck. By default we capture only 20 levels of stack back trace on suspension

**HPX_MAX_NETWORK_RETRIES**

**HPX_NETWORK_RETRIES_SLEEP**

**HPX_INI_PATH_DELIMITER**

**HPX_PATH_DELIMITERS**

796 Chapter 2. What’s so special about HPX?
HPX_SHARED_LIB_EXTENSION
HPX_EXECUTABLE_EXTENSION
HPX_MAKE_DLL_STRING \(n\)
HPX_MANGLE_NAME \(n\)
HPX_MANGLE_STRING \(n\)
HPX_COMPONENT_NAME_DEFAULT
HPX_COMPONENT_NAME
HPX_COMPONENT_STRING
HPX_PLUGIN_COMPONENT_PREFIX
HPX_PLUGIN_NAME_DEFAULT
HPX_PLUGIN_NAME
HPX_PLUGIN_STRING
HPX_PLUGIN_PLUGIN_PREFIX
HPX_IDLE_LOOP_COUNT_MAX
HPX_BUSY_LOOP_COUNT_MAX
HPX_THREAD_QUEUE_MAX_THREAD_COUNT
HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_PENDING
HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_STAGED
HPX_THREAD_QUEUE_MIN_ADD_NEW_COUNT
HPX_THREAD_QUEUE_MAX_ADD_NEW_COUNT
HPX_THREAD_QUEUE_MIN_DELETE_COUNT
HPX_THREAD_QUEUE_MAX_DELETE_COUNT
HPX_THREAD_QUEUE_MAX_TERMINATED_THREADS
HPX_THREAD_QUEUE_INIT_THREADS_COUNT
HPX_IDLE_BACKOFF_TIME_MAX
HPX_WRAPPER_HEAP_STEP
HPX_INITIAL_GID_RANGE
HPX_CONTINUATION_MAX_RECURSION_DEPTH

Defines

HPX_NOINLINE
Function attribute to tell compiler not to inline the function.

HPX_NORETURN
Function attribute to tell compiler that the function does not return.

HPX_DEPRECATED \(x\)
Marks an entity as deprecated. The argument \(x\) specifies a custom message that is included in the compiler warning. For more details see \(<>\).
**HPX_FALLTHROUGH**
Indicates that the fall through from the previous case label is intentional and should not be diagnosed by a compiler that warns on fallthrough. For more details see `<>__`.

**HPX_NODISCARD**
If a function declared nodiscard or a function returning an enumeration or class declared nodiscard by value is called from a discarded-value expression other than a cast to void, the compiler is encouraged to issue a warning. For more details see `__`.

**HPX_NO_UNIQUE_ADDRESS**
Indicates that this data member need not have an address distinct from all other non-static data members of its class. For more details see `__`.

**Defines**

**HPX_LIKELY**(expr)
Hint at the compiler that expr is likely to be true.

**HPX_UNLIKELY**(expr)
Hint at the compiler that expr is likely to be false.

**Defines**

**HPX_COMPILER_FENCE**
Generates assembly that serves as a fence to the compiler CPU to disable optimization. Usually implemented in the form of a memory barrier.

**HPX_SMT_PAUSE**
Generates assembly the executes a “pause” instruction. Useful in spinning loops.

**Defines**

**HPX_NATIVE_TLS**
This macro is replaced with the compiler specific keyword attribute to mark a variable as thread local. For more details see `__`.

This macro is deprecated. It is always replaced with the thread_local keyword. Prefer using thread_local directly instead.

**Defines**

**HPX_GCC_VERSION**
Returns the GCC version HPX is compiled with. Only set if compiled with GCC.

**HPX_CLANG_VERSION**
Returns the Clang version HPX is compiled with. Only set if compiled with Clang.

**HPX_INTEL_VERSION**
Returns the Intel Compiler version HPX is compiled with. Only set if compiled with the Intel Compiler.

**HPX_MSVC**
This macro is set if the compilation is with MSVC.

**HPX_MINGW**
This macro is set if the compilation is with Mingw.
HPX_WINDOWS
This macro is set if the compilation is for Windows.

HPX_NATIVE_MIC
This macro is set if the compilation is for Intel Knights Landing.

Defines

HPX_CONSTEXPR
This macro evaluates to constexpr if the compiler supports it.

This macro is deprecated. It is always replaced with the constexpr keyword. Prefer using constexpr directly instead.

HPX_CONSTEXPR_OR_CONST
This macro evaluates to constexpr if the compiler supports it, const otherwise.

This macro is deprecated. It is always replaced with the constexpr keyword. Prefer using constexpr directly instead.

HPX_INLINE_CONSTEXPR_VARIABLE
This macro evaluates to inline constexpr for host code.

HPX_HOST_DEVICE_INLINE_CONSTEXPR_VARIABLE
This macro evaluates to inline constexpr for host code and.

HPX_STATIC_CONSTEXPR
This macro evaluates to static constexpr if the compiler supports it, static const otherwise.

This macro is deprecated. It is always replaced with the static constexpr keyword. Prefer using static constexpr directly instead.

Defines

HPX_DEBUG
Defined if HPX is compiled in debug mode.

HPX_BUILD_TYPE
Evaluates to debug if compiled in debug mode, release otherwise.
**HPX_DEPRECATED_V** (*major, minor, x*)

Defines

**HPX_NON_COPYABLE** (*cls*)

Marks a class as non-copyable and non-movable.

Defines

**HPX_EXPORT**

Marks a class or function to be exported from HPX or imported if it is consumed.

Defines

**HPX_FORCEINLINE**

Marks a function to be forced inline.

Defines

**HPX_CAPTURE_FORWARD** (*var*)

Evaluates to `var = std::forward<decltype(var)>(var)` if the compiler supports C++14 Lambdas. Defaults to `var`.

This macro is deprecated. Prefer using `var = std::forward<decltype(var)>(var)` directly instead.

**HPX_CAPTURE_MOVE** (*var*)

Evaluates to `var = std::move(var)` if the compiler supports C++14 Lambdas. Defaults to `var`.

This macro is deprecated. Prefer using `var = std::move(var)` directly instead.

Defines

**HPX_CXX20_CAPTURE_THIS** (...)

Defines

**HPX_SUPER_PURE**

**HPX_PURE**

**HPX_HOT**

**HPX_COLD**

800 Chapter 2. What’s so special about **HPX**?
Defines

HPX_THREADS.Stack.Overhead
HPX.Small.Stack.Size
HPX.Medium.Stack.Size
HPX.Large.Stack.Size
HPX.Huge.Stack.Size

Defines

HPX.Weak.Symbol

config_registry

The contents of this module can be included with the header `hpx/modules/config_registry.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/config_registry.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

HPX_CONFIG_REGISTRY_EXPORT
namespace hpx

namespace config_registry

Functions

HPX_CONFIG_REGISTRY_EXPORT std::vector<module_config> const& hpx::config_registry::get_module_configs()
HPX_CONFIG_REGISTRY_EXPORT void hpx::config_registry::add_module_config(module_config const & config)
struct add_module_config_helper

Public Functions

add_module_config_helper(module_config const & config)
struct module_config
Public Members

std::string module_name
std::vector<std::string> config_entries

coroutines

The contents of this module can be included with the header hpx/modules/coroutines.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/coroutines.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace threads

namespace coroutines

class coroutine

Public Types

using impl_type = detail::coroutine_impl
using thread_id_type = impl_type::thread_id_type
using result_type = impl_type::result_type
using arg_type = impl_type::arg_type
using functor_type = util::unique_function_nonser<result_type (arg_type)>

Public Functions

coroutine (functor_type &&f, thread_id_type id, std::ptrdiff_t stack_size = detail::default_stack_size)
coroutine (coroutine const &src)
coroutine &operator= (coroutine const &src)
coroutine (coroutine &&src)
coroutine &operator= (coroutine &&src)
thread_id_type get_thread_id () const
std::size_t get_thread_data () const
std::size_t set_thread_data (std::size_t data)
void init ()
void rebinding (function_type &&f, thread_id_type id)
result_type operator() (arg_type arg = arg_type())
bool is_ready () const

t::ptrdiff_t get_available_stack_space()
impl_type *impl()

Private Members

impl_type impl_

namespace hpx

namespace threads

namespace coroutines

class stackless_coroutine

Public Types

using thread_id_type = hpx::threads::thread_id
using result_type = std::pair<thread_schedule_state, thread_id_type>
using arg_type = thread_restart_state
using functor_type = util::unique_function_nonser<result_type (arg_type)>

Public Functions

stackless_coroutine (function_type &&f, thread_id_type id, std::ptrdiff_t = default_stack_size)
~stackless_coroutine ()
stackless_coroutine (stackless_coroutine const &src)
stackless_coroutine &operator= (stackless_coroutine const &src)
stackless_coroutine (stackless_coroutine &&src)
stackless_coroutine &operator= (stackless_coroutine &&src)
thread_id_type get_thread_id () const
std::size_t get_thread_data () const
std::size_t set_thread_data (std::size_t data)
void rebinding (function_type &&f, thread_id_type id)
void reset_tss()
void reset()

stackless_coroutine::result_type operator() (arg_type arg = arg_type())

operator bool() const
bool is_ready() const
std::ptrdiff_t get_available_stack_space()
std::size_t &get_continuation_recursion_count()

Protected Attributes

functor_type f_
context_state state_
thread_id_type id_
std::size_t thread_data_
std::size_t continuation_recursion_count_

Private Types

enum context_state

Values:

ctx_running
ctx_ready
ctx_exited

Private Functions

bool running() const
bool exited() const

Private Static Attributes

constexpr std::ptrdiff_t default_stack_size = -1
Friends

friend hpx::threads::coroutines::reset_on_exit
struct reset_on_exit

Public Functions

reset_on_exit (stackless_coroutine &this__)
~reset_on_exit ()

Public Members

stackless_coroutine &this__

Defines

HPX_THREAD_STATE_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_PRIORITY_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_STATE_EX_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_STACKSIZE_UNSCOPED_ENUM_DEPRECATED_MSG
HPX_THREAD_SCHEDULE_HINT_UNSCOPED_ENUM_DEPRECATED_MSG
namespace hpx

namespace threads

Enums

define thread_schedule_state
The thread_schedule_state enumerator encodes the current state of a thread instance

Values:
unknown = 0
active = 1
thread is currently active (running, has resources)
pending = 2
thread is pending (ready to run, but no hardware resource available)
suspended = 3
thread has been suspended (waiting for synchronization event, but still known and under control
of the thread-manager)
depleted = 4
thread has been depleted (deeply suspended, it is not known to the thread-manager)
terminated = 5
thread has been stopped an may be garbage collected

2.8. API reference
staged = 6
this is not a real thread state, but allows to reference staged task descriptions, which eventually
will be converted into thread objects

pending_do_not_schedule = 7
pending_boost = 8

datastruct thread

This enumeration lists all possible thread-priorities for HPX threads.

Values:

unknown = -1

default = 0
Will assign the priority of the task to the default (normal) priority.

low = 1
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.

normal = 2
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.

high_recursive = 3
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.

boost = 4
Same as thread_priority_high except that the thread will fall back to thread_priority_normal if
resumed after being suspended.

high = 5
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).

bound = 6
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.

datastruct thread_restart_state

The thread_restart_state enumerator encodes the reason why a thread is being restarted

Values:

unknown = 0

signaled = 1
The thread has been signaled.

timeout = 2
The thread has been reactivated after a timeout

terminate = 3
The thread needs to be terminated.

abort = 4
The thread needs to be aborted.

datastruct thread_stacksize

A thread_stacksize references any of the possible stack-sizes for HPX threads.
Values:

unknown = -1
small_ = 1
  use small stack size (the underscore is to work around small being defined to char on Windows)
medium = 2
  use medium sized stack size
large = 3
  use large stack size
huge = 4
  use very large stack size
nostack = 5
  this thread does not suspend (does not need a stack)
current = 6
  use size of current thread’s stack
default_ = small_
  use default stack size
minimal = small_
  use minimally stack size
maximal = huge
  use maximally stack size

enum thread_schedule_hint_mode
  The type of hint given when creating new tasks.

Values:

none = 0
  A hint that leaves the choice of scheduling entirely up to the scheduler.

thread = 1
  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.

numa = 2
  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.
**Functions**

```cpp
std::ostream &operator<< (std::ostream &os, thread_schedule_state const t)
```

char const *get_thread_state_name (thread_schedule_state state)

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.

```cpp
std::ostream &operator<< (std::ostream &os, thread_priority const t)
```

char const *get_thread_priority_name (thread_priority priority)

Return the thread priority name.

Get the readable string representing the name of the given thread_priority constant.

**Parameters**

- **this**: represents the thread priority.

```cpp
std::ostream &operator<< (std::ostream &os, thread_restart_state const t)
```

char const *get_thread_state_ex_name (thread_restart_state state)

Get the readable string representing the name of the given thread_restart_state constant.

char const *get_thread_state_name (thread_state state)

Get the readable string representing the name of the given thread_state constant.

```cpp
std::ostream &operator<< (std::ostream &os, thread_stacksize const t)
```

char const *get_stack_size_enum_name (thread_stacksize size)

Returns the stack size name.

Get the readable string representing the given stack size constant.

**Parameters**

- **size**: this represents the stack size

---

**Variables**

```cpp
constexpr thread_schedule_state unknown = thread_schedule_state::unknown
constexpr thread_schedule_state active = thread_schedule_state::active
constexpr thread_schedule_state pending = thread_schedule_state::pending
constexpr thread_schedule_state suspended = thread_schedule_state::suspended
constexpr thread_schedule_state depleted = thread_schedule_state::depleted
constexpr thread_schedule_state terminated = thread_schedule_state::terminated
constexpr thread_schedule_state staged = thread_schedule_state::staged
constexpr thread_schedule_state pending_do_not_schedule = thread_schedule_state::pending_do_not_schedule
```
constexpr thread_schedule_state pending_boost = thread_schedule_state::pending_boost
constexpr thread_priority thread_priority_unknown = thread_priority::unknown
constexpr thread_priority thread_priority_default = thread_priority::default_
constexpr thread_priority thread_priority_low = thread_priority::low
constexpr thread_priority thread_priority_normal = thread_priority::normal
constexpr thread_priority thread_priority_high_recursive = thread_priority::high_recursive
constexpr thread_priority thread_priority_boost = thread_priority::boost
constexpr thread_priority thread_priority_high = thread_priority::high
constexpr thread_priority thread_priority_bound = thread_priority::bound
constexpr thread_priority thread_priority_critical = thread_priority::critical
constexpr thread_restart_state wait_unknown = thread_restart_state::unknown
constexpr thread_restart_state wait_signaled = thread_restart_state::signaled
constexpr thread_restart_state wait_timeout = thread_restart_state::timeout
constexpr thread_restart_state wait_terminate = thread_restart_state::terminate
constexpr thread_restart_state wait_abort = thread_restart_state::abort
constexpr thread_stacksize thread_stacksize_unknown = thread_stacksize::unknown
constexpr thread_stacksize thread_stacksize_small = thread_stacksize::small_
constexpr thread_stacksize thread_stacksize_medium = thread_stacksize::medium
constexpr thread_stacksize thread_stacksize_large = thread_stacksize::large
constexpr thread_stacksize thread_stacksize_huge = thread_stacksize::huge
constexpr thread_stacksize thread_stacksize_nostack = thread_stacksize::nostack
constexpr thread_stacksize thread_stacksize_current = thread_stacksize::current
constexpr thread_stacksize thread_stacksize_default = thread_stacksize::default_
constexpr thread_stacksize thread_stacksize_minimal = thread_stacksize::minimal
constexpr thread_stacksize thread_stacksize_maximal = thread_stacksize::maximal

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

constexpr thread_schedule_hint ()
Construct a default hint with mode thread_schedule_hint_mode::none.

constexpr thread_schedule_hint (std::int16_t thread_hint)
Construct a hint with mode thread_schedule_hint_mode::thread and the given hint as the local thread number.
constexpr thread_schedule_hint (thread_schedule_hint_mode mode, std::int16_t hint)

Construct a hint with the given mode and hint. The numerical hint is unused when the mode is thread_schedule_hint_mode::none.

Public Members

std::int16_t hint

The hint associated with the mode. The interpretation of this hint depends on the given mode.

thread_schedule_hint_mode mode

The mode of the scheduling hint.

Public Functions

std::size_t operator() (::hpx::threads::thread_id const &v) const

Public Functions

std::size_t operator() (::hpx::threads::thread_id_ref const &v) const

namespace hpx

namespace threads

Enums

define thread_id_addref

Values:

yes

no

Variables

HPX_INLINE_CONSTEXPR_VARIABLE const thread_id hpx::threads::invalid_thread_id

struct thread_id
Public Functions

thread_id()

thread_id(thread_id const&)

thread_id &operator=(thread_id const&)

cconstexpr thread_id(thread_id &&rhs)

cconstexpr thread_id &operator=(thread_id &&rhs)

cconstexpr thread_id (thread_id_repr const &thrd)

cconstexpr thread_id &operator=(thread_id_repr const &rhs)

cconstexpr thread_id_repr get () const

cconstexpr void reset ()

Private Types

using thread_id_repr = void*

Private Members

thread_id_repr thrd_ = nullptr

Friends

friend constexpr bool operator==(std::nullptr_t, thread_id const &rhs)

friend constexpr bool operator!=(std::nullptr_t, thread_id const &rhs)

friend constexpr bool operator==(thread_id const &lhs, std::nullptr_t)

friend constexpr bool operator!=(thread_id const &lhs, std::nullptr_t)

friend constexpr bool operator==(thread_id const &lhs, thread_id const &rhs)

friend constexpr bool operator!=(thread_id const &lhs, thread_id const &rhs)

friend constexpr bool operator<(thread_id const &lhs, thread_id const &rhs)

friend constexpr bool operator>(thread_id const &lhs, thread_id const &rhs)

friend constexpr bool operator<=(thread_id const &lhs, thread_id const &rhs)

friend constexpr bool operator>=(thread_id const &lhs, thread_id const &rhs)

template<typename Char, typename Traits>

void format_value (std::ostream &os, boost::string_ref spec, thread_id const &id)

struct thread_id_ref
Public Types

using thread_repr = detail::thread_data_reference_counting

Public Functions

thread_id_ref()

thread_id_ref(thread_id_ref const&)
thread_id_ref& operator=(thread_id_ref const&)

thread_id_ref(thread_id_ref &&rhs)
thread_id_ref& operator=(thread_id_ref &&rhs)

thread_id_ref(thread_id_repr const &thrd)

thread_id_ref(thread_id_repr &&thrd)
thread_id_ref& operator=(thread_id_repr const &rhs)

thread_id_ref(thread_id_repr &&thrd)

thread_id_ref(thread_id *thrd, thread_id_addref addref = thread_id_addref::yes)
thread_id_ref& operator=(thread_id_repr *rhs)

thread_id_ref(thread_id const &noref)

thread_id_ref(thread_id &&noref)
thread_id_ref& operator=(thread_id const &noref)

thread_id_ref& operator=(thread_id &&noref)

operator bool() const

thread_id noref() const

thread_id_repr& get() &

thread_id_repr& get() &&

thread_id_repr const &get() const &

void reset()

Private Types

using thread_id_repr = hpx::intrusive_ptr<detail::thread_data_reference_counting>
Private Members

thread_id_repr thrd_

Friends

bool operator== (std::nullptr_t, thread_id_ref const &rhs)
bool operator!= (std::nullptr_t, thread_id_ref const &rhs)
bool operator== (thread_id_ref const &lhs, std::nullptr_t)
bool operator!= (thread_id_ref const &lhs, std::nullptr_t)
bool operator== (thread_id_ref const &lhs, thread_id_ref const &rhs)
bool operator!= (thread_id_ref const &lhs, thread_id_ref const &rhs)
bool operator< (thread_id_ref const &lhs, thread_id_ref const &rhs)
bool operator> (thread_id_ref const &lhs, thread_id_ref const &rhs)
bool operator<== (thread_id_ref const &lhs, thread_id_ref const &rhs)
bool operator==> (thread_id_ref const &lhs, thread_id_ref const &rhs)

template<typename Char, typename Traits>
std::basic_ostream<Char, Traits> &operator<< (std::basic_ostream<Char, Traits> &os, thread_id_ref const &id)
void format_value (std::ostream &os, boost::string_ref spec, thread_id_ref const &id)

namespace std

template<>
struct hash<::hpx::threads::thread_id>

Public Functions

std::size_t operator() (::hpx::threads::thread_id const &v) const

template<>
struct hash<::hpx::threads::thread_id_ref>

Public Functions

std::size_t operator() (::hpx::threads::thread_id_ref const &v) const
datastructures

The contents of this module can be included with the header hpx/modules/datastructures.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/datastructures.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

template<class>
class basic_any<void, void, void, std::true_type>

Public Functions

constexpr basic_any ()
basic_any (basic_any const &x)
basic_any (basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any (T &&&x, typename std::enable_if<!std::is_copy_constructible<typename std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value &&std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
basic_any (std::in_place_type_t<T>, Ts&&... ts)

~basic_any ()
basic_any &operator= (basic_any const &x)
basic_any &operator= (basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any &operator= (T &&&rhs)
basic_any &swap (basic_any &x)

std::type_info const &type () const

template<typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const &rhs) const
Private Functions

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 void new_object (void *object, std::true_type, Ts&&... ts)
template<typename T, typename ...Ts>
static void new_object (void *object, std::false_type, Ts&&... ts)
template<typename Char>
class basic_any<Char, std::true_type>

Public Functions

constexpr basic_any ()

basic_any (basic_any const &x)

basic_any (basic_any &&x)

template<typename T, typename Enable=typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any (T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable=typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts&&... ts>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
basic_any (std::in_place_type_t<T>, std::initializer_list<T> il, Ts&&... ts)

~basic_any ()

basic_any &operator= (basic_any const &x)

basic_any &operator= (basic_any &&rhs)

template<typename T, typename Enable=typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any &operator= (T &&rhs)

basic_any &swap (basic_any &x)

std::type_info const &type () const

template<typename T>
T const &cast () const

bool has_value () const
void reset ()

bool equal_to (basic_any const & rhs) const

Private Functions

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 void new_object (void *& object, std::true_type, Ts&&... ts)

template<typename T, typename ... Ts>
static void new_object (void *& object, std::false_type, Ts&&... ts)

template<>

class basic_any<void, void, void, std::false_type>

Public Functions

constexpr basic_any ()

basic_any (basic_any && x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename T::type>::value>::type>
basic_any (T && x, typename T::type std::enable_if<!std::is_move_constructible<typename T::type>::value>::type * = nullptr)

template<typename T, typename ... Ts, typename Enable = typename std::enable_if<std::is_constructible<typename T::type, Ts&&... ts>::value>::type>
basic_any (std::in_place_type_t<T>, Ts&&... ts)

template<typename T, typename U, typename ... Ts, typename Enable = typename std::enable_if<std::is_constructible<typename T::type, std::initializer_list<U> && il, Ts&&... ts>::value>::type>
basic_any (std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

basic_any (basic_any const & x)

basic_any & operator= (basic_any const & x)

~basic_any ()

basic_any & operator= (basic_any && rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename T::type>::value>::type>
basic_any & operator= (T && rhs)

basic_any & swap (basic_any & x)

std::type_info const & type () const
template<typename T>
T const &cast() const

bool has_value() const

void reset()

bool equal_to(basic_any const &rhs) const

**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 void new_object(void *object, std::true_type, Ts &&... ts)

template<typename T, typename ...Ts>
static void new_object(void *object, std::false_type, Ts &&... ts)

template<typename Char>
class basic_any<void, void, Char, std::false_type>

**Public Functions**

castexpr basic_any()

basic_any(basic_any &&&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any(T &&&x, typename std::enable_if<std::is_move_constructible<typename std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts &&... ts>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
basic_any(std::in_place_type_t<T>, Ts &&... ts)

basic_any(basic_any const &&x)

basic_any &operator=(basic_any const &&x)

~basic_any()

basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any &operator=(T &&rhs)

basic_any &swap(basic_any &&x)

std::type_info const &type() const
template<typename T>
    T const &cast() const

bool has_value() const

void reset()

bool equal_to(basic_any const &rhs) const

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 void new_object(void *object, std::true_type, Ts&& ...ts)

template<typename T, typename ...Ts>
static void new_object(void *object, std::false_type, Ts&& ...ts)

namespace hpx

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, std::false_type> make_any_nonser(std::initializer_list<U> il, Ts&& ...ts)

template<typename T, typename ...Ts>
util::basic_any<void, void, std::false_type> make_unique_any_nonser(Ts&& ...ts)

template<typename T, typename U, typename ...Ts>
util::basic_any<void, void, std::false_type> make_unique_any_nonser(std::initializer_list<U> il, Ts&& ...ts)

template<typename T>
util::basic_any<void, void, std::true_type> make_any_nonser(T &t)

template<typename T>
util::basic_any<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)
template<
type T, typename IArch, typename OArch, typename Char, typename Copyable>
T const *any_cast (util::basic_any<IArch, OArch, Char, Copyable> const *operand)

template<
type T, typename IArch, typename OArch, typename Char, typename Copyable>
T any_cast (util::basic_any<IArch, OArch, Char, Copyable> &operand)

template<
type T, typename IArch, typename OArch, typename Char, typename Copyable>
T const &any_cast (util::basic_any<IArch, OArch, Char, Copyable> const &operand)

struct bad_any_cast : public bad_cast

Public Functions

bad_any_cast (std::type_info const &src, std::type_info const &dest)

const char *what () const

Public Members

const char *from

const char *to

namespace util

Typedefs

typedef hpx::tuple_element<I, T> instead

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<
type IArch, typename OArch, typename Char, typename Copyable, typename Enable = typename
std::basic_istream<Char> &operator>>(std::basic_istream<Char> &i, basic_any<IArch, OArch,
Char, Copyable> &obj)

template<
type IArch, typename OArch, typename Char, typename Copyable, typename Enable = typename
std::basic_ostream<Char> &operator<<(std::basic_ostream<Char> &o, basic_any<IArch, OArch,
Char, Copyable> const &obj)

template<
type IArch, typename OArch, typename Char, typename Copyable>
void swap (basic_any<IArch, OArch, Char, Copyable> &lhs, basic_any<IArch, OArch, Char, Copy-
able> &rhs)

template<
type T, typename... Ts>
hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_any_nonser is deprecated. Please use "
hpx::util::make_any_nonser(Ts&&... ts)

template<
type T, typename U, typename... Ts>
hpx::util::HPX_DEPRECATED_V(1, 6, "
std::true_type make_any_nonser (std::initializer_list<U> il, Ts&&... ts)

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... Ts>
hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_unique_any_nonser is deprecated. Please use hpx::make_unique_any_nonser instead.")

std::false_type make_unique_any_nonser (Ts&&... ts)

template< typename T, typename U, typename... Ts>
hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_unique_any_nonser is deprecated. Please use hpx::make_unique_any_nonser instead.")

std::false_type make_unique_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>
hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_any_nonser is deprecated. Please use hpx::make_unique_any_nonser instead.")

std::true_type make_any_nonser (T &&t)

template< typename T, typename Char>
basic_any<void, void, Char, std::true_type> make_streamable_any_nonser (T &&t)

template< typename T>
hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::make_unique_any_nonser is deprecated. Please use hpx::make_unique_any_nonser instead.")

std::false_type make_unique_any_nonser (T &&t)

template< typename T, typename Char>
basic_any<void, void, Char, std::false_type> make_streamable_unique_any_nonser (T &&t)

Variables

hpx::util::void

template<typename Char>
class basic_any<void, void, Char, std::false_type>

Public Functions

constexpr basic_any ()

basic_any (basic_any &&&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<typename std::decay<T>::type>::value>::value* = nullptr)
template<typename T, typename ...Ts, typename Enable = typename std::enable_if<!std::is_same<decltype(T)::type, typename std::decay<T>::type>::value && std::is_move_constructible<typename std::decay<T>::type>::value>::type>

basic_any (basic_any const &x)
basic_any &operator=(basic_any const &x)
~basic_any ()
basic_any &operator=(basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<decltype(T)::type, typename std::decay<T>::type>::value && std::is_move_constructible<typename std::decay<T>::type>::value>::type>

basic_any &operator=(T &&rhs)
basic_any &swap (basic_any &x)

std::type_info const &type() const

template<typename T>

T const &cast() const

bool has_value() const

void reset()

bool equal_to (basic_any const &rhs) const

**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 void new_object (void *object, std::true_type, Ts&&... ts)

template<typename T, typename ...Ts>

static void new_object (void *object, std::false_type, Ts&&... ts)

template<typename Char>

class basic_any<void, void, Char, std::true_type>

2.8. API reference
Public Functions

```cpp
consexpr basic_any()

basic_any(basic_any const &x)

basic_any(basic_any &&x)
```

template<
typename T,
typename Enable = std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value, typename std::decay<T>::type>::value* = nullptr

```cpp
basic_any(T &&, typename std::enable_if<!std::is_copy_constructible<typename std::decay<T>::type>::value>::type*)
```

```cpp
template<typename Ts, typename Enable = std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>

basic_any(std::in_place_type_t<T>, Ts&&... ts)
```

```cpp
~basic_any()
```

basic_any &operator=(basic_any const &x)

basic_any &operator=(basic_any &&rhs)

template<
typename T,
typename Enable = std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value, typename std::decay<T>::type>::value>

```cpp
basic_any &operator=(T &&rhs)
```

```cpp
basic_any &swap(basic_any &x)
```

```cpp
std::type_info const &type() const
```

template<
typename T>

```cpp
T const &cast() const
```

bool has_value() const

void reset()

bool equal_to(basic_any const &rhs) const

Private Functions

basic_any &assign(basic_any const &x)

Private Members

detail::fxn_ptr_table<void, void, Char, std::true_type> *table

void *object
Private Static Functions

template<typename T, typename ...Ts>
static void new_object (void * & object, std::true_type, Ts &&... ts)

template<typename T, typename ...Ts>
static void new_object (void * & object, std::false_type, Ts &&... ts)

template<>
class basic_any< void, void, void, std::false_type>

Public Functions

constexpr basic_any ()

basic_any (basic_any & & x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any (T & & x, typename std::enable_if<std::is_move_constructible<typename std::decay<T>::type>::value>::type* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::type, Ts...>::value && std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
basic_any (std::in_place_type_t<T>, Ts &&... ts)

basic_any (basic_any const & x)

basic_any & operator= (basic_any const & x)

~basic_any ()

basic_any & operator= (basic_any & & rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any & operator= (T & & rhs)

basic_any & swap (basic_any & x)

std::type_info const & type () const

template<typename T>
T const & cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const & rhs) const
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 void new_object (void *object, std::true_type, Ts&&... ts)
template< typename T, typename ...Ts>
static void new_object (void *object, std::false_type, Ts&&... ts)

template<>
class basic_any <void, void, void, std::true_type>

Public Functions

constexpr basic_any ()

basic_any (basic_any const &x)

basic_any (basic_any &&x)

template< typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value>::type>

basic_any(T &&x, typename std::enable_if<std::is_copy_constructible<typename std::decay<T>::value>::value>::type* = nullptr)

template< typename T, typename ...Ts, typename Enable = typename std::enable_if<std::is_constructible<typename std::decay<T>::value, Ts&&... ts>::value &&std::is_copy_constructible<typename std::decay<T>::value>::value>::type>

basic_any (std::in_place_type_t<T>, std::initializer_list<U> il, Ts&&... ts)

~basic_any ()

basic_any &operator= (basic_any const &x)

basic_any &operator= (basic_any &&rhs)

template< typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value>::type>

basic_any &operator= (T &&rhs)

basic_any &swap (basic_any &x)

std::type_info const &type () const

template< typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const &rhs) const
Private Functions

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 void new_object (void *object, std::true_type, Ts&&... ts)
template<typename T, typename ...Ts>
static void new_object (void *object, std::false_type, Ts&&... ts)

template<std::size_t... Is, typename ...Ts>
struct member_pack<util::index_pack<Is...>, Ts...> : public hpx::util::detail::member_leaf<Is, Ts>

Public Functions

member_pack ()

template<typename ...Us>
constexpr member_pack (std::piecewise_construct_t, Us&&... us)

template<std::size_t I>
constexpr decltype(auto) get () &

template<std::size_t I>
constexpr decltype(auto) get () const &

template<std::size_t I>
constexpr decltype(auto) get () &&

template<std::size_t I>
constexpr decltype(auto) get () const &&

namespace hpx

namespace serialization
Functions

template<typename Archive, std::size_t... Is, typename... Ts>
void serialize(Archive & ar, ::hpx::util::member_pack<util::index_pack<Is...>, Ts...>& mp, unsigned int const = 0)

namespace util

Typedefs

template<typename... Ts>
using member_pack_for = member_pack<
    typename util::make_index_pack<sizeof...(Ts)>::type,
    Ts...>

Variables

template<typename Is, typename... Ts>
struct HPX_EMPTY_BASES member_pack

template<std::size_t... Is, typename... Ts>
struct member_pack<
    util::index_pack<Is...>, Ts...>
    : public hpx::util::detail::member_leaf<Is, Ts>

Public Functions

    member_pack()

    template<typename... Us>
    constexpr member_pack(std::piecewise_construct_t, Us&&... us)

    template<std::size_t I>
    constexpr decltype(auto) get() &

    template<std::size_t I>
    constexpr decltype(auto) get() const &

    template<std::size_t I>
    constexpr decltype(auto) get() &&

    template<typename T>
    struct hash<hpx::optional<T>>

Public Functions

    constexpr std::size_t operator() (::hpx::optional<T> const &arg) const

namespace hpx

namespace util
Functions

template<typename T>
constexpr bool operator==(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator!=(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator<(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator>=(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator>(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator<=(optional<T> const &lhs, optional<T> const &rhs)

template<typename T>
constexpr bool operator==(optional<T> const &opt, nullopt_t)

template<typename T>
constexpr bool operator==(nullopt_t, optional<T> const &opt)

template<typename T>
constexpr bool operator!=(optional<T> const &opt, nullopt_t)

template<typename T>
constexpr bool operator!=(nullopt_t, optional<T> const &opt)

template<typename T>
constexpr bool operator<(optional<T> const&, nullopt_t)

template<typename T>
constexpr bool operator<(nullopt_t, optional<T> const&)

template<typename T>
constexpr bool operator>=(optional<T> const&, nullopt_t)

template<typename T>
constexpr bool operator>=(nullopt_t, optional<T> const&)

template<typename T>
constexpr bool operator>(optional<T> const& nullopt_t)

template<typename T>
constexpr bool operator>(nullopt_t, optional<T> const&)

template<typename T>
constexpr bool operator<=(optional<T> const& nullopt_t)

template<typename T>
constexpr bool operator<=(nullopt_t, optional<T> const&)

template<typename T>
constexpr bool operator==(optional<T> const &opt, T const &value)
template<typename T>
constexpr bool operator== (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator!= (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator!= (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator< (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator< (T const &value, optional<T> const &opt)

template<typename T>
constexpr bool operator<= (optional<T> const &opt, T const &value)

template<typename T>
constexpr bool operator<= (T const &value, optional<T> const &opt)

template<typename T>
void swap (optional<T> &x, optional<T> &y)

template<typename T>
constexpr optional<typename std::decay<T>::type> make_optional (T &&v)

template<typename T, typename ...Ts>
constexpr optional<T> make_optional (Ts&&... ts)

template<typename T, typename U, typename ...Ts>
constexpr optional<T> make_optional (std::initializer_list<U> il, Ts&&... ts)

Variables

cconstexpr nullopt_t nullopt = {nullopt_t::init()}

class bad_optional_access : public logic_error
Public Functions

bad_optional_access (std::string const &what_arg)
bad_optional_access (char const *what_arg)

struct nullopt_t

Public Functions

constexpr nullopt_t (nullopt_t::init)

template<typename T>
class optional

Public Types

template<>
using value_type = T

Public Functions

constexpr optional ()
constexpr optional (nullopt_t)
optional (optional const &other)
optional (optional &other)
optional (T const &val)
optional (T &&val)

template<typename ...Ts>
optional (in_place_t, Ts&&... ts)

template<typename U, typename ...Ts>
optional (in_place_t, std::initializer_list<U> il, Ts&&... ts)

~optional ()
optional &operator= (optional const &other)
optional &operator= (optional &&other)
optional &operator= (T const &other)
optional &operator= (T &&other)
optional &operator= (nullopt_t)
constexpr T const *operator-> () const
T *operator-> ()
constexpr T const &operator*() const
T &operator*()

constexpr operator bool() const
constexpr bool has_value() const
T &value()
T const &value() const

template<typename U>
constexpr T value_or(U &&value) const

template<typename ...Ts>
void emplace(Ts&&... ts)

template<typename F, typename ...Ts>
void emplace_f(F &&f, Ts&&... ts)

void swap(optional &other)
void reset()

Private Members

std::aligned_storage<sizeof(T), alignof(T)>::type storage_
bool empty_

namespace _optional_swap

Functions

template<typename T>
void check_swap()
Defines

\texttt{HPX\_DEFINE\_TAG\_SPECIFIER(\texttt{NAME})}

namespace hpx

namespace util

template<typename \texttt{Base}, typename ...\texttt{Tags}>
struct tagged

Public Functions

template<typename ...\texttt{Ts}>
tagged (\texttt{T}s&& ... \texttt{ts})

template<typename \texttt{Other}>
tagged (tagged<\texttt{Other}, Tags...> && \texttt{rhs})

template<typename \texttt{Other}>
tagged (tagged<\texttt{Other}, Tags...> \texttt{const} & \texttt{rhs})

template<typename \texttt{Other}>
tagged &\texttt{operator=} (tagged<\texttt{Other}, Tags...> && \texttt{rhs})

tagged &\texttt{operator=} (tagged<\texttt{Other}, Tags...> \texttt{const} & \texttt{rhs})

tagged &\texttt{operator=} (\texttt{U} && \texttt{u})

void\texttt{swap} (tagged &\texttt{other})

Friends

void\texttt{swap} (tagged &\texttt{x}, tagged &\texttt{y})

namespace hpx

namespace util

Functions

template<typename \texttt{Tag1}, typename \texttt{Tag2}, typename \texttt{T1}, typename \texttt{T2}>
constexpr tagged_pair<\texttt{Tag1} (typename \texttt{std}::decay<\texttt{T1}>::type) , \texttt{Tag2} (typename \texttt{std}::decay<\texttt{T2}>::type) \texttt{make\_tagged\_pair}\texttt{std}::pair<\texttt{T1}, \texttt{T2}> &&\texttt{p}

template<typename \texttt{Tag1}, typename \texttt{Tag2}, typename \texttt{T1}, typename \texttt{T2}>
constexpr tagged_pair<\texttt{Tag1} (typename \texttt{std}::decay<\texttt{T1}>::type) , \texttt{Tag2} (typename \texttt{std}::decay<\texttt{T2}>::type) \texttt{make\_tagged\_pair}\texttt{std}::pair<\texttt{T1}, \texttt{T2}> \texttt{const} &\texttt{p}

template<typename \texttt{Tag1}, typename \texttt{Tag2}, typename ...\texttt{Ts}>

2.8. API reference 831
constexpr tagged_pair<Tag1 (typename hpx::tuple_element<0, hpx::tuple<Ts...>::type), Tag2
    typename hpx::tuple_element<1, hpx::tuple<Ts...>::type> make_tagged_pair hpx::tuple<Ts...>

&&p

template< typename Tag1, typename Tag2, typename ...Ts>
constexpr tagged_pair<Tag1 (typename hpx::tuple_element<0, hpx::tuple<Ts...>::type), Tag2
    typename hpx::tuple_element<1, hpx::tuple<Ts...>::type> make_tagged_pair hpx::tuple<Ts...>
    const &p

template< typename Tag1, typename Tag2, typename T1, typename T2>
constexpr tagged_pair<Tag1 (typename std::decay<T1>::type), Tag2
    typename std::decay<T2>::type> make_tagged_pair T1 &&&t1, T2 &&&t2

template< typename F, typename S>
struct tagged_pair: public hpx::util::tagged<std::pair<typename detail::tag_elem<F>::type, typename detail::tag_elem<S>::type>,

Public Types

typedef tagged<std::pair<typename detail::tag_elem<F>::type, typename detail::tag_elem<S>::type>, typename
type, typename
type>

Public Functions

template< typename ...Ts>
tagged_pair (Ts&&... ts)

namespace hpx

namespace util

Functions

template< typename ...Tags, typename ...Ts>
constexpr tagged_tuple<typename detail::tagged_type<Tags, Ts>::type...> make_tagged_tuple (Ts&&... ts)

template< typename ...Tags, typename ...Ts>
constexpr tagged_tuple<typename detail::tagged_type<Tags, Ts>::type...> make_tagged_tuple (hpx::tuple<Ts...> &&&t)

template< typename ...Ts>
struct tagged_tuple : public hpx::util::tagged<hpx::tuple<typename detail::tag_elem<Ts>::type...>, typename detail::tag_spec<Ts>::type...>,

Public Types

template<>
using base_type = tagged<hpx::tuple<typename detail::tag_elem<Ts>::type...>, typename detail::tag_spec<Ts>::type...>
Public Functions

template<typename... Ts_>
tagged_tuple (Ts_&&... ts)
template<typename T0, typename T1>
struct tuple_element<0, std::pair<T0, T1>>

Public Types

template<>
using type = T0

Public Static Functions

static constexpr type & get (std::pair<T0, T1> &tuple)
static constexpr type const & get (std::pair<T0, T1> const &tuple)
template<typename T0, typename T1>
struct tuple_element<1, std::pair<T0, T1>>

Public Types

template<>
using type = T1

Public Static Functions

static constexpr type & get (std::pair<T0, T1> &tuple)
static constexpr type const & get (std::pair<T0, T1> const &tuple)
template<std::size_t I, typename Type, std::size_t Size>
struct tuple_element<I, std::array<Type, Size>>

Public Types

template<>
using type = Type

Public Static Functions

static constexpr type & get (std::array<Type, Size> &tuple)
static constexpr type const & get (std::array<Type, Size> const &tuple)

namespace hpx
Functions

```cpp
template<typename ...Ts>
constexpr tuple<typename util::decay_unwrap<Ts>::type...> make_tuple (Ts&&... vs)

template<typename ...Ts>
tuple<Ts&&...> forward_as_tuple (Ts&&... vs)

template<typename ...Ts>
tuple<Ts&&...> tie (Ts&&... vs)

template<typename ...Tuples>
constexpr auto tuple_cat (Tuples&&... tuples)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator== (tuple<Ts...> const &t, tuple<Us...> const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator!= (tuple<Ts...> const &t, tuple<Us...> const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator< (tuple<Ts...> const &t, tuple<Us...> const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator> (tuple<Ts...> const &t, tuple<Us...> const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator<= (tuple<Ts...> const &t, tuple<Us...> const &u)

template<typename ...Ts, typename ...Us>
constexpr std::enable_if<sizeof...(Ts) == sizeof...(Us), bool>::type operator>= (tuple<Ts...> const &t, tuple<Us...> const &u)

void swap (tuple<Ts...> &x, tuple<Ts...> &y)
```
Variables

\texttt{const hpx::detail::ignore\_type ignore = {}}

template<typename ...Ts>
\texttt{class tuple}

\textbf{Public Functions}

\texttt{template<typename Dependent = void, typename Enable = typename std::enable\_if<util::all\_of<std::is\_constructible<Ts>>>::value, constexpr tuple ()}

\texttt{constexpr tuple (Ts const\&... vs)}

\texttt{template<typename U, typename ...Us, typename Enable = typename std::enable\_if<!std::is\_same<tuple, typename std::decay<U>::type>::value || util::pack<Us...>::size != 0>::type, typename EnableCompatible = typename std::enable\_if<hpx::detail::are\_tuples\_compatible<tuple, tuple<U, Us...>>::value>::type> constexpr tuple (U &&v, Us &&... vs)}

\texttt{tuple (tuple const\&)}

\texttt{tuple (tuple &&)}

\texttt{template<typename UTuple, typename Enable = typename std::enable\_if<!std::is\_same<tuple, typename std::decay<UTuple>::type>::value>::type, typename EnableCompatible = typename std::enable\_if<hpx::detail::are\_tuples\_compatible<tuple, UTuple>::value>::type> constexpr tuple (UTuple &&other)}

\texttt{tuple &operator= (tuple const &other)}

\texttt{tuple &operator= (tuple &&& other)}

\texttt{template<typename UTuple> tuple &operator= (UTuple &&& other)}

\texttt{void swap (tuple &other)}

\texttt{template<util::size\_t I> util::at\_index<I, Ts...>::type &get ()}

\texttt{template<util::size\_t I> util::at\_index<I, Ts...>::type const &get () const}

\textbf{Private Types}

\texttt{template<>}

\texttt{using index\_pack = typename util::make\_index\_pack<sizeof...(Ts)>::type}

\textbf{Private Functions}

\texttt{template<util::size\_t... Is, typename UTuple> constexpr tuple (util::index\_pack<Is...>, UTuple &&&other)}

\texttt{template<util::size\_t... Is> void assign_ (util::index\_pack<Is...>, tuple const &other)}

\texttt{template<util::size\_t... Is> void assign_ (util::index\_pack<Is...>, tuple &&&other)}
template< std::size_t ... Is, typename UTuple >
void assign_( util::index_pack<Is...>, UTuple &other )

template< std::size_t ... Is >
void swap_ ( util::index_pack<Is...>, tuple &other )

Private Members

util::member_pack forall< Ts... >_members

template<>
class tuple<>

Public Functions

constexpr tuple ()
cconstexpr tuple ( tuple const& )
cconstexpr tuple ( tuple&& )
tuple &operator= ( tuple const& )
tuple &operator= ( tuple&& )
void swap ( tuple& )

template< typename T0, typename T1 >
struct tuple_element<0, std::pair<T0, T1>>

Public Types

template<>
using type = T0

Public Static Functions

static constexpr type & get ( std::pair<T0, T1> &tuple )
static constexpr type const & get ( std::pair<T0, T1> const &tuple )

template< typename T0, typename T1 >
struct tuple_element<1, std::pair<T0, T1>>
**Public Types**

```cpp
template<>
using type = T1
```

**Public Static Functions**

```cpp
static constexpr type & get (std::pair<T0, T1> & tuple)
static constexpr type const & get (std::pair<T0, T1> const & tuple)
```

```cpp
template<std::size_t I, typename Type, std::size_t Size>
struct tuple_element<I, std::array<Type, Size>>
```

**Public Types**

```cpp
template<>
using type = Type
```

**Public Static Functions**

```cpp
static constexpr type & get (std::array<Type, Size> & tuple)
static constexpr type const & get (std::array<Type, Size> const & tuple)
```

```cpp
template<std::size_t I, typename ...Ts>
struct tuple_element<I, tuple<Ts...>>
```

**Public Types**

```cpp
template<>
using type = typename util::at_index::type
```

**Public Static Functions**

```cpp
static constexpr type & get (hpx::tuple<Ts...> & tuple)
static constexpr type const & get (hpx::tuple<Ts...> const & tuple)
```

```cpp
template<class T>
struct tuple_size
    Subclassed by hpx::tuple_size< const T >, hpx::tuple_size< const volatile T >, hpx::tuple_size< volatile T >
```

```cpp
namespace adl_barrier
```
Functions

```cpp
template<
  std::size_t I,
  typename Tuple,
  typename Enable = typename std::always_void<
    typename hpx::tuple_element<I, Tuple>::type &
    (Tuple &t)
  >::type

constexpr hpx::tuple_element<I, Tuple>::type &
  (Tuple &t)

template<
  std::size_t I,
  typename Tuple,
  typename Enable = typename std::always_void<
    typename hpx::tuple_element<I, Tuple>::type constexpr &
    &
    (Tuple &t)
  >::type

constexpr hpx::tuple_element<I, Tuple>::type constexpr &
  & (Tuple &t)

template<
  std::size_t I,
  typename Tuple,
  typename Enable = typename std::always_void<
    typename hpx::tuple_element<I, Tuple>::type const &
    &
    (Tuple &t)
  >::type

constexpr hpx::tuple_element<I, Tuple>::type constexpr &
  & (Tuple &t)

template<
  std::size_t I,
  typename Tuple,
  typename Enable

constexpr tuple_element<I, Tuple>::type & (Tuple &t)

template<
  std::size_t I,
  typename Tuple,
  typename Enable

constexpr tuple_element<I, Tuple>::type constexpr & (Tuple &t)

template<
  std::size_t I,
  typename Tuple,
  typename Enable

constexpr tuple_element<I, Tuple>::type constexpr & (Tuple &t)

template<
  std::size_t I,
  typename Tuple,
  typename Enable

constexpr tuple_element<I, Tuple>::type constexpr & (Tuple &t)

namespace std_adl_barrier

Functions

```cpp
template<
  std::size_t I,
  typename ... Ts>

constexpr hpx::tuple_element<I, hpx::tuple<Ts...>>::type &
  (hpx::tuple<Ts...> & t)

template<
  std::size_t I,
  typename ... Ts>

constexpr hpx::tuple_element<I, hpx::tuple<Ts...>>::type constexpr &
  & (hpx::tuple<Ts...> & & t)

template<
  std::size_t I,
  typename ... Ts>

constexpr hpx::tuple_element<I, hpx::tuple<Ts...>>::type constexpr &
  & (hpx::tuple<Ts...> const & t)

template<
  std::size_t I,
  typename ... Ts>

constexpr tuple_element<I, tuple<Ts...>>::type &
  (tuple<Ts...> & t)

template<
  std::size_t I,
  typename ... Ts>

constexpr tuple_element<I, tuple<Ts...>>::type constexpr &
  & (tuple<Ts...> const & t)

template<
  std::size_t I,
  typename ... Ts>

constexpr tuple_element<I, tuple<Ts...>>::type constexpr &
  & (tuple<Ts...> & t)

template<
  std::size_t I,
  typename ... Ts>

constexpr tuple_element<I, tuple<Ts...>>::type constexpr &
  & (tuple<Ts...> const & t)

namespace util
Functions

`std::HPX_DEPRECATED_V(1, 6, "hpx::util::ignore is deprecated. Use hpx::ignore instead.")

`std::HPX_DEPRECATED_V(1, 6, "hpx::util::make_tuple is deprecated. Use hpx::make_tuple instead."

`std::HPX_DEPRECATED_V(1, 6, "hpx::util::tie is deprecated. Use hpx::tie instead."

`std::HPX_DEPRECATED_V(1, 6, "hpx::util::forward_as_tuple is deprecated. Use hpx::forward_as_tuple instead."

`std::HPX_DEPRECATED_V(1, 6, "hpx::util::tuple_cat is deprecated. Use hpx::tuple_cat instead."

namespace hpx

namespace adl_barrier

Functions

`std::size_t I, typename ...Ts>`
`constexpr hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::get is deprecated. Use hpx::get instead.") const & get (std::variant<Ts...> & var)

`std::size_t I, typename ...Ts>`
`constexpr hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::get is deprecated. Use hpx::get instead.") const && get (std::variant<Ts...> && var)

`std::size_t I, typename ...Ts>`
`constexpr hpx::util::HPX_DEPRECATED_V(1, 6, "hpx::util::get is deprecated. Use hpx::get instead.") const & get (std::variant<Ts...> const & var)

namespace hpx

namespace traits

`template<typename T>`
`struct is_tuple_like : public hpx::traits::detail::is_tuple_like_impl<std::remove_cv<T>::type>

#include <is_tuple_like.hpp>` Deduces to a true type if the given parameter T has a specific tuple like size.

debugging

The contents of this module can be included with the header hpx/modules/debugging.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/debugging.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace util
Functions

void attach_debugger()
Tries to break an attached debugger, if not supported a loop is invoked which gives enough time to attach a debugger manually.

namespace hpx

namespace util

Functions

std::string trace(std::size_t frames_no = HPX_HAVE_THREAD_BACKTRACE_DEPTH)

namespace hpx

namespace util

namespace debug

Typedefs

template<typename T>
using cxxabi_demangle_helper = demangle_helper<T>

template<typename T>
using cxx_type_id = type_id<T>

Functions

template<typename T = void>
std::string print_type(const char* = "")

template<>
std::string print_type(const char* delim)

template<typename T>
struct demangle_helper

Public Functions

char const *type_id() const

template<typename T>
struct type_id
Public Static Attributes

demangle_helper<T> typeid_ = demangle_helper<T>();

Variables

char **environ

Defines

HPX_DP.LAZY (Expr, printer)

namespace hpx

namespace util

Functions

template<typename E>
details::trace_manip trace(E const &e)

namespace details

Functions

std::ostream &operator<<(std::ostream &out, details::trace_manip const &t)

class trace_manip

Public Functions

trace_manip (backtrace const *tr)

std::ostream &write (std::ostream &out) const

Private Members

backtrace const *tr_

namespace stack_trace
Functions

`std::size_t trace(void **addresses, std::size_t size)`

`void write_symbols(void *const *addresses, std::size_t size, std::ostream&)`

`std::string get_symbol(void *address)`

`std::string get_symbols(void *const *address, std::size_t size)`

errors

The contents of this module can be included with the header `hpx/modules/errors.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we *strongly* suggest only including the module header `hpx/modules/errors.hpp`, not the particular header in which the functionality you would like to use is defined. See *Public API* for a list of names that are part of the public HPX API.

namespace hpx

 Enums

 enum error
 Possible error conditions.

 This enumeration lists all possible error conditions which can be reported from any of the API functions.

 Values:

 `success` = 0
 The operation was successful.

 `no_success` = 1
 The operation did failed, but not in an unexpected manner.

 `not_implemented` = 2
 The operation is not implemented.

 `out_of_memory` = 3
 The operation caused an out of memory condition.

 `bad_action_code` = 4

 `bad_component_type` = 5
 The specified component type is not known or otherwise invalid.

 `network_error` = 6
 A generic network error occurred.

 `version_too_new` = 7
 The version of the network representation for this object is too new.

 `version_too_old` = 8
 The version of the network representation for this object is too old.

 `version_unknown` = 9
 The version of the network representation for this object is unknown.

 `unknown_component_address` = 10
duplicate_component_address = 11
   The given global id has already been registered.
invalid_status = 12
   The operation was executed in an invalid status.
bad_parameter = 13
   One of the supplied parameters is invalid.
internal_server_error = 14
service_unavailable = 15
bad_request = 16
repeated_request = 17
lock_error = 18
duplicate_console = 19
   There is more than one console locality.
no_registered_console = 20
   There is no registered console locality available.
startup_timed_out = 21
uninitialized_value = 22
bad_response_type = 23
deadlock = 24
assertion_failure = 25
null_thread_id = 26
   Attempt to invoke a API function from a non-HPX thread.
invalid_data = 27
yield_aborted = 28
   The yield operation was aborted.
dynamic_link_failure = 29
commandline_option_error = 30
   One of the options given on the command line is erroneous.
serialization_error = 31
   There was an error during serialization of this object.
unhandled_exception = 32
   An unhandled exception has been caught.
kernel_error = 33
   The OS kernel reported an error.
broken_task = 34
   The task associated with this future object is not available anymore.
task_moved = 35
   The task associated with this future object has been moved.
task_already_started = 36
   The task associated with this future object has already been started.
future_already_retrieved = 37
The future object has already been retrieved.

promise_already_satisfied = 38
The value for this future object has already been set.

future_does_not_support_cancellation = 39
The future object does not support cancellation.

future_can_not_be_cancelled = 40
The future can’t be canceled at this time.

no_state = 41
The future object has no valid shared state.

broken.promise = 42
The promise has been deleted.

thread_resource_error = 43

future_cancelled = 44

thread_cancelled = 45

thread_not_interruptable = 46

duplicate_component_id = 47
The component type has already been registered.

unknown_error = 48
An unknown error occurred.

bad_plugin_type = 49
The specified plugin type is not known or otherwise invalid.

filesystem_error = 50
The specified file does not exist or other filesystem related error.

bad_function_call = 51
equivalent of std::bad_function_call

task_canceled_exception = 52
parallel::v2::task_canceled_exception

task_block_not_active = 53
task_region is not active

out_of_range = 54
Equivalent to std::out_of_range.

length_error = 55
Equivalent to std::length_error.

migration_needs_retry = 56
migration failed because of global race, retry

namespace hpx
error_code make_error_code (error e, throwmode mode = plain)
    Returns a new error_code constructed from the given parameters.

error_code make_error_code (error e, char const *func, char const *file, long line, throwmode mode = plain)
error_code make_error_code (error e, char const *msg, throwmode mode = plain)
    Returns error_code(e, msg, mode).

error_code make_error_code (error e, char const *msg, char const *func, char const *file, long line, throwmode mode = plain)
error_code make_error_code (error e, std::string const &msg, throwmode mode = plain)
    Returns error_code(e, msg, mode).

error_code make_error_code (error e, std::string const &msg, char const *func, char const *file, long line, throwmode mode = plain)
error_code make_error_code (std::exception_ptr const &e)

Functions

std::error_category const &get_hpx_category ()
    Returns generic HPX error category used for new errors.

std::error_category const &get_hpx_rethrow_category ()
    Returns generic HPX error category used for errors re-thrown after the exception has been de-serialized.

error_code make_success_code (throwmode mode = plain)
    Returns error_code(hpx::success, “success”, mode).

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

error_code (throwmode mode = 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).

Exceptions
• nothing:
error_code\( (error \ e, \ \text{throwmode} \ \text{mode} = \text{plain}) \)
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).

Exceptions
- nothing:

error_code\( (error \ e, \ \text{const} \ \text{func}, \ \text{const} \ \text{file}, \ \text{long} \ \text{line}, \ \text{throwmode} \ \text{mode} = \text{plain}) \)
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).

Exceptions
- nothing:

error_code\( (error \ e, \ \text{const} \ \text{msg}, \ \text{throwmode} \ \text{mode} = \text{plain}) \)
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).

Exceptions
- std::bad_alloc: (if allocation of a copy of the passed string fails).

error_code\( (error \ e, \ \text{const} \ \text{msg}, \ \text{const} \ \text{func}, \ \text{const} \ \text{file}, \ \text{long} \ \text{line}, \ \text{throwmode} \ \text{mode} = \text{plain}) \)
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).

Exceptions
- std::bad_alloc: (if allocation of a copy of the passed string fails).
**error_code** (*error e, std::string const &msg, throwmode mode = 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`).

**Exceptions**
- `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 = 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`).

**Exceptions**
- `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`.

**Exceptions**
- `nothing`:

void **clear**()

Clear this `error_code` object. The postconditions of invoking this method are.

- `value() == hpx::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

\texttt{error\_code (int err, hpx::exception const &e)}

\texttt{error\_code (std::exception\_ptr const &e)}

Private Members

\texttt{std::exception\_ptr exception_}

Friends

\texttt{friend hpx::exception error\_code make\_error\_code (std::exception\_ptr const &e)}

\texttt{namespace hpx}

Typedefs

\texttt{using custom\_exception\_info\_handler\_type = std::function<hpx::exception\_info (std::string const&, std::string const&, long, std::string const&)>}

\texttt{using pre\_exception\_handler\_type = std::function<void ()>}

Functions

\texttt{void set\_custom\_exception\_info\_handler (custom\_exception\_info\_handler\_type f)}

\texttt{void set\_pre\_exception\_handler (pre\_exception\_handler\_type f)}

\texttt{std::string get\_error\_what (exception\_info const &xi)}

Return The error message of the thrown exception.

The function \texttt{hpx::get\_error\_what} can be used to extract the diagnostic information element representing the error message as stored in the given exception instance.

Parameters

848 Chapter 2. What’s so special about \textit{HPX}?
• 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.

Exceptions
• std::bad_alloc: (if one of the required allocations fails)

error get_error (hpx::exception const &e)
Return the error value code 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.

Return 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.

See 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.

Exceptions
• nothing:

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.

Return 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.

See 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_backtrace(), hpx::get_error_env(), hpx::get_error_what(), hpx::get_error_config(), hpx::get_error_state()

Parameters
• xi: The parameter xi 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.

Exceptions
• std::bad_alloc: (if one of the required allocations fails)
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.

**Return** 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.

**See** 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_backtrace(), hpx::get_error_env(), hpx::get_error_what(), hpx::get_error_config(), hpx::get_error_thread()

**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.

**Exceptions**

- std::bad_alloc: (if one of the required allocations fails)

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.

**Return** 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.

**See** 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_backtrace(), hpx::get_error_env(), hpx::get_error_what(), hpx::get_error_config(), hpx::get_error_thread()

**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.

**Exceptions**

- nothing:

```
#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
```

850 Chapter 2. What’s so special about HPX?
Public Functions

**exception** (error e = success)
Construct a `hpx::exception` from a `hpx::error`.

**Parameters**
- `e`: The parameter `e` holds the `hpx::error` code the new exception should encapsulate.

**exception** (std::system_error const &e)
Construct a `hpx::exception` from a boost::system_error.

**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 = 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 = 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** ()
Destruct a `hpx::exception`

**Exceptions**
- nothing:

**error** get_error() const
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.

**Exceptions**
- nothing:

**error_code** get_error_code (throwmode mode = plain) const
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 `plain`, this is the default) or to the category
  `hpx_category_rethrow` (if mode is `rethrow`).

`struct thread_interrupted : public exception`

```cpp
#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.

```cpp
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 re-
versed 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.

```cpp
void g()
{
    // 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
    }
}
```

(continues on next page)
At any point, the interruption state for the current thread can be queried by calling `hpx::this_thread::interruption_enabled()`.

### Enums

**enum throwmode**

Encode error category for new `error_code`.

Values:

- **plain** = 0
- **rethrow** = 1
- **lightweight** = 0x80

### Variables

**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.

### Defines

```cpp
HPX_DEFINE_ERROR_INFO(NAME, TYPE)
```

namespace hpx
Functions

```cpp
template<typename E> HPX_NORETURN void hpx::throw_with_info(E && e, exception_info && xi)
template<typename E> HPX_NORETURN void hpx::throw_with_info(E && e, exception_info const & xi)
```

```cpp
template<typename E> exception_info *get_exception_info(E & e)
template<typename E> exception_info const *get_exception_info(E const & e)
```

```cpp
template<typename E, typename F>
auto invoke_with_exception_info(E const & e, F && f)
template<typename F>
auto invoke_with_exception_info(std::exception_ptr const & p, F && f)
template<typename F>
auto invoke_with_exception_info(hpx::error_code const & ec, F && f)
```

```cpp
template<typename Tag, typename Type>
struct error_info
```

Public Types

```cpp
template<> using tag = Tag
template<> using type = Type
```

Public Functions

```cpp
error_info (Type const & value)
error_info (Type && value)
```

Public Members

```cpp
Type _value
```

class exception_info
Subclassed by hpx::detail::exception_with_info_base
Public Functions

exception_info()

exception_info(exception_info const &other)

exception_info(exception_info &&other)

exception_info &operator=(exception_info const &other)

exception_info &operator=(exception_info &&other)

virtual ~exception_info()

template<typename ...ErrorInfo>
exception_info &set(ErrorInfo&&... tagged_values)

template<typename Tag>
Tag::type const *get() const

Private Types

using node_ptr = std::shared_ptr<detail::exception_info_node_base>

Private Members

node_ptr _data

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

std::size_t size() const
The number of exception_ptr objects contained within the exception_list.

Note Complexity: Constant time.

exception_list_type::const_iterator begin() const
An iterator referring to the first exception_ptr object contained within the exception_list.

exception_list_type::const_iterator end() const
An iterator which is the past-the-end value for the exception_list.
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.

```cpp
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::no_success, "raise_exception", "simulated error");
}
```

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.

**execution**

The contents of this module can be included with the header `hpx/modules/execution.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/execution.hpp`, not the particular header in which the functionality you would like to use is defined. See **Public API** for a list of names that are part of the public `HPX` API.

```cpp
namespace hpx
{
    namespace execution
    {
        namespace experimental
        {
        }
    }
}
```
Variables

`hpx::execution::experimental::bulk_t bulk`

```
struct bulk_t : public hpx::functional::detail::tag_priority<bulk_t>
```

Friends

```template<typename Sender, typename Shape, typename F>
friend constexpr auto tag_override_invoke (bulk_t, Sender &&sender, Shape const &shape, F &&f)
```

```template<typename Sender, typename Shape, typename F>
friend constexpr auto tag_fallback_invoke (bulk_t, Sender &&&sender, Shape const &shape, F &&f)
```

```template<typename Sender, typename Shape, typename F>
friend constexpr auto tag_fallback_invoke (bulk_t, Sender &&&sender, Shape &&shape, F &&f)
```

```template<typename Shape, typename F>
friend constexpr auto tag_fallback_invoke (bulk_t, Shape &&&shape, F &&f)
```

namespace hpx

```namespace execution```

```namespace experimental```

Variables

`hpx::execution::experimental::detach_t detach`

```struct detach_t : public hpx::functional::detail::tag_fallback<detach_t>
```

Friends

```template<typename Sender, typename Allocator = hpx::util::internal_allocator<>>
friend constexpr void tag_fallback_invoke (detach_t, Sender &&&sender, Allocator const &allocator = Allocator{})
```

```template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke (detach_t, Allocator const &allocator = Allocator{})
```
Variables

*hpx::execution::experimental::ensure_started_t* **ensure_started**

```cpp
struct ensure_started_t : public hpx::functional::detail::tag_fallback<ensure_started_t>
```

**Friends**

```cpp
template<typename Sender, typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(ensure_started_t, Sender &&sender, Allocator const &allocator = {})
```

```cpp
template<typename Sender, typename Allocator>
friend constexpr auto tag_fallback_invoke(ensure_started_t, detail::split_sender<Sender, Allocator, detail::submission_type::eager> sender, Allocator const & = {})  
```

```cpp
template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(ensure_started_t, Allocator const &allocator = {})
```

**namespace hpx**

**namespace execution**

**namespace experimental**

**Variables**

*hpx::execution::experimental::execute_t* **execute**

```cpp
struct execute_t : public hpx::functional::detail::tag_fallback<execute_t>
```

**Friends**

```cpp
template<typename Scheduler, typename F>
friend constexpr auto tag_fallback_invoke(execute_t, Scheduler &&scheduler, F &&f)
```

**namespace hpx**

**namespace execution**

**namespace experimental**
Variables

HPX_INLINE_CONSTEXPR_VARIABLE struct hpx::execution::experimental::just_t hpx::execution::experimental::just

struct just_t

Public Functions

template<typename ...Ts>
constexpr auto operator() (Ts&&... ts) const

namespace hpx

namespace execution

namespace experimental

Variables

hpx::execution::experimental::just_on_t just_on

struct just_on_t : public hpx::functional::detail::tag_fallback<just_on_t>

Friends

template<typename Scheduler, typename ...Ts>
friend constexpr auto tag_fallback_invoke (just_on_t, Scheduler &&scheduler, Ts&&... ts)

namespace hpx

namespace execution

namespace experimental

Variables

HPX_INLINE_CONSTEXPR_VARIABLE struct hpx::execution::experimental::keep_future_t hpx::execution::experimental::keep_future

struct keep_future_t
Public Functions

template<typename Future>
constexpr auto operator() (Future &future) const
constexpr auto operator() () const

namespace hpx

namespace execution

namespace experimental

Variables

hpx::execution::experimental::let_error_t let_error

struct let_error_t : public hpx::functional::detail::tag_fallback<let_error_t>

Friends

template<typename PredecessorSender, typename F>
friend constexpr auto tag_fallback_invoke (let_error_t, PredecessorSender &&predecessor_sender, F &&f)

template<typename F>
friend constexpr auto tag_fallback_invoke (let_error_t, F &&f)

namespace hpx

namespace execution

namespace experimental

Variables

hpx::execution::experimental::let_value_t let_value

struct let_value_t : public hpx::functional::detail::tag_fallback<let_value_t>

Friends

template<typename PredecessorSender, typename F>
friend constexpr auto tag_fallback_invoke (let_value_t, PredecessorSender &&predecessor_sender, F &&f)

template<typename F>
friend constexpr auto tag_fallback_invoke (let_value_t, F &&f)
namespace execution

namespace experimental

Variables

`hpx::execution::experimental::make_future_t make_future`

`struct make_future_t : public hpx::functional::detail::tag_fallback<make_future_t>`

Friends

```cpp
template<typename Sender, typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(make_future_t, Sender &&sender, Allocator const &allocator = Allocator{})
```

```cpp
template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(make_future_t, Allocator const &allocator = Allocator{})
```

namespace hpx

namespace execution

namespace experimental

Variables

`hpx::execution::experimental::on_t on`

`struct on_t : public hpx::functional::detail::tag_fallback<on_t>`

Friends

```cpp
template<typename Sender, typename Scheduler>
friend constexpr auto tag_fallback_invoke(on_t, Sender &&predecessor_sender, Scheduler &&scheduler)
```

```cpp
template<typename Scheduler>
friend constexpr auto tag_fallback_invoke(on_t, Scheduler &&scheduler)
```

namespace hpx

namespace execution

namespace experimental
Variables

`hpx::execution::experimental::split_t split`

```
struct split_t : public hpx::functional::detail::tag_fallback<split_t>
```

Friends

```
template<
    typename Sender,
    typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(split_t, Sender &&sender,
    Allocator const &allocator = {}) {
}
```

```
template<
    typename Sender,
    typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(split_t, detail::split_sender<Sender,
    Allocator, submission_type::lazy> sender, Allocator const &allocator = {}) {
}
```

```
template<typename Allocator = hpx::util::internal_allocator<>>
friend constexpr auto tag_fallback_invoke(split_t, Allocator const &allocator = {}) {
}
```

```namespace hpx```
```
namespace execution```
```
namespace experimental```
```

Variables

`hpx::execution::experimental::sync_wait_t sync_wait`

```
struct sync_wait_t : public hpx::functional::detail::tag_fallback<sync_wait_t>
```

Friends

```
template<
    typename Sender>
friend constexpr auto tag_fallback_invoke(sync_wait_t, Sender &&sender) {
}
```

```
friend constexpr auto tag_fallback_invoke(sync_wait_t) {
}
```

```namespace hpx```
```
namespace execution```
```
namespace experimental```
Variables

\texttt{hpx::execution::experimental::transform\_t transform}

\textbf{struct transform\_t : public hpx::functional::detail::tag\_fallback<transform\_t>}

Friends

\begin{verbatim}
template<typename Sender, typename F>
friend constexpr auto tag\_fallback\_invoke (transform\_t, Sender &&sender, F &&f)

template<typename F>
friend constexpr auto tag\_fallback\_invoke (transform\_t, F &&f)
\end{verbatim}

namespace hpx

namespace execution

namespace experimental

Variables

\texttt{hpx::execution::experimental::when\_all\_t when\_all}

\textbf{struct when\_all\_t : public hpx::functional::detail::tag\_fallback<when\_all\_t>}

Friends

\begin{verbatim}
template<typename ...Senders>
friend constexpr auto tag\_fallback\_invoke (when\_all\_t, Senders&&... senders)
\end{verbatim}

namespace hpx

namespace execution

\textbf{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

**constexpr auto_chunk_size**

`constexpr auto_chunk_size(std::uint64_t num_iters_for_timing = 0)`

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.

```cpp
auto_chunk_size(hpx::chrono::steady_duration const &rel_time, std::uint64_t num_iters_for_timing = 0)
```

Construct an 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.

namespace parallel

namespace execution

**Typedefs**

```cpp
typedef hpx::is_sequenced_execution_policy<T> instead
```

namespace hpx

namespace execution

**struct dynamic_chunk_size**

```cpp
#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

**constexpr dynamic_chunk_size**

`constexpr dynamic_chunk_size(std::size_t chunk_size = 1)`

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.
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()

Parameters
• exec: [in] The executor object to use for querying the number of pending tasks.
• topo: [in] The topology object to use to extract the requested information.
• thread_num: [in] The sequence number of the thread to retrieve information for.

Private Functions

template<typename Executor>
dectype(auto) friend tag_fallback_invoke (get pu mask_t, Executor&&,
threads::topology &topo, std::size_t thread_num)

template<typename Executor>
dectype(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

Parameters
• exec: [in] The executor object to use to extract the requested information for.

Private Functions

template<typename Executor>
dectype(auto) friend tag_fallback_invoke (has pending closures_t, Executor&&)

template<typename Executor>
dectype(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 under-
neath the given executor.

Note This calls exec.set scheduler_mode(mode) if it exists; otherwise it does nothing.
Parameters

- **exec**: [in] The executor object to use.
- **mode**: [in] The new mode for the scheduler to pick up

**Friends**

template<typename Executor, typename Mode>
void tag_fallback_invoke (set_scheduler_mode_t, Executor&&, Mode const&)  

template<typename Executor, typename Mode>
void tag_invoke (set_scheduler_mode_t, Executor &&exec, Mode const &mode)

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)

template<typename ...Params>
struct executor_parameters_join

**Public Types**

template<>
using type = detail::executor_parameters<std::decay_t<Params>...>

template<typename Param>
struct executor_parameters_join<Param>

**Public Types**

template<>
using type = Param

namespace hpx

namespace parallel

namespace execution
Variables

hpx::parallel::execution::get_chunk_size_t get_chunk_size
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::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_fallback<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

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).

Parameters

- params: [in] The executor parameters object to use for determining the chunk size for the given number of tasks num_tasks.
- exec: [in] The executor object which will be used for scheduling of the loop iterations.
- f: [in] The function which will be optionally scheduled using the given executor.
- cores: [in] The number of cores the number of chunks should be determined for.
- num_tasks: [in] The number of tasks the chunk size should be determined for

Private Functions

template<typename Parameters, typename Executor, typename F>
dectype(auto) friend tag_fallback_invoke (get_chunk_size_t, Parameters &&params, Executor &&exec, F &&f, std::size_t cores, std::size_t num_tasks)

struct mark_begin_execution_t : public hpx::functional::detail::tag_fallback<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.

Parameters

- params: [in] The executor parameters object to use as a fallback if the executor does not expose
Private Functions

```cpp
template<typename Parameters, typename Executor>
decay_type(auto) friend_tag_fallback_invoke(mark_begin_execution_t, Parameters &&params, Executor &&exec);
```

```cpp
struct mark_end_execution_t : public hpx::functional::detail::tag_fallback<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.

**Parameters**
- **params**: [in] The executor parameters object to use as a fallback if the executor does not expose

---

Private Functions

```cpp
template<typename Parameters, typename Executor>
decay_type(auto) friend_tag_fallback_invoke(mark_end_of_scheduling_t, Parameters &&params, Executor &&exec);
```

```cpp
struct mark_end_of_scheduling_t : public hpx::functional::detail::tag_fallback<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.

**Parameters**
- **params**: [in] The executor parameters object to use as a fallback if the executor does not expose

---

Private Functions

```cpp
template<typename Parameters, typename Executor>
decay_type(auto) friend_tag_fallback_invoke(maximal_number_of_chunks_t, Parameters &&params, Executor &&exec);
```

```cpp
struct maximal_number_of_chunks_t : public hpx::functional::detail::tag_fallback<maximal_number_of_chunks_t>
#include <execution_parameters_fwd.hpp> Return the largest reasonable number of chunks to create for a single algorithm invocation.
```

**Parameters**
- **params**: [in] The executor parameters object to use for determining the number of chunks for the given number of cores.
- **exec**: [in] The executor object which will be used for scheduling of the loop iterations.
- **cores**: [in] The number of cores the number of chunks should be determined for.
- **num_tasks**: [in] The number of tasks the chunk size should be determined for.
Private Functions

```cpp
template<typename Parameters, typename Executor>
decayt(auto) friend tag fallback invoke(maximal_number_of_chunks_t, Parameters &&params, Executor &&exec, std::size_t cores, std::size_t num_tasks)
```

```cpp
struct processing_units_count_t : public hpx::functional::detail::tag fallback<processing_units_count_t>
```

```cpp
#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.

**Parameters**
- `params`: [in] The executor parameters object to use as a fallback if the executor does not expose

Private Functions

```cpp
template<typename Parameters, typename Executor>
decayt(auto) friend tag fallback invoke(processing_units_count_t, Parameters &&params, Executor &&exec)
```

```cpp
struct reset_thread_distribution_t : public hpx::functional::detail::tag fallback<reset_thread_distribution_t>
```

```cpp
#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.

**Parameters**
- `params`: [in] The executor parameters object to use for resetting the thread distribution scheme.
- `exec`: [in] The executor object to use.

Private Functions

```cpp
namespace hpx
```

```cpp
namespace execution
```

```cpp
struct guided_chunk_size
```

```cpp
#include <guided_chunk_size.hpp>
```

Iterations are dynamically assigned to threads in blocks as threads request them 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

```cpp
constexpr guided_chunk_size (std::size_t min_chunk_size = 1)
```

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.

```cpp
namespace hpx

namespace execution

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.

Public Functions

```cpp
constexpr persistent_auto_chunk_size (std::uint64_t num_iters_for_timing = 0)
```
Construct a 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.

```cpp
persistent_auto_chunk_size (hpx::chrono::steady_duration const &time_cs,
                         std::uint64_t num_iters_for_timing = 0)
```
Construct a persistent_auto_chunk_size executor parameters object

Parameters

- `time_cs`: The execution time for each chunk.

```cpp
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)
```
Construct a 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.
namespace parallel

namespace execution

template<typename R, typename ...Ts>
class polymorphic_executor<R(Ts...)>
  private
    hpx::parallel::execution::detail::polymorphic_executor_base

Public Types

template<typename T>
using future_type = hpx::future<R>

Public Functions

castexpr polymorphic_executor ()
polymorphic_executor (polymorphic_executor const &other)
polymorphic_executor (polymorphic_executor &&other)
polymorphic_executor &operator= (polymorphic_executor const &other)
polymorphic_executor &operator= (polymorphic_executor &&other)
template<typename Exec, typename PE = typename std::decay<Exec>::type, typename Enable = typename polymorphic_executor (Exec &&exec)
template<typename Exec, typename PE = typename std::decay<Exec>::type, typename Enable = typename polymorphic_executor (Exec &exec)
void reset ()
template<typename F>
void post (F &&f, Ts... ts) const

template<typename F>
R sync_execute (F &&f, Ts... ts) const

template<typename F>
hpx::future<R> async_execute (F &&f, Ts... ts) const

template<typename F, typename Future>
hpx::future<R> then_execute (F &&f, Future &&predecessor, Ts&&... ts) const

template<typename F, typename Shape>
std::vector<R> bulk_sync_execute (F &&f, Shape const &s, Ts&&... ts) const

template<typename F, typename Shape>
std::vector<hpx::future<R>> bulk_async_execute (F &&f, Shape const &s, Ts&&... ts) const

template<typename F, typename Shape>
hpx::future<std::vector<R>> bulk_then_execute (F &&f, Shape const &s, hpx::shared_future<void> const &predecessor, Ts&&... ts) const
Private Types

template<>
using base_type = detail::polymorphic_executor_base
template<>
using vtable = detail::polymorphic_executor_vtable<R(Ts...)>

Private Functions

void assign (std::nullptr_t)
template<typename Exec>
void assign (Exec &&exec)

Private Static Functions

static constexpr vtable const *get_empty_vtable()
template<typename T>
static constexpr vtable const *get_vtable()

namespace hpx

namespace parallel

namespace execution

Variables

HPX_INLINE_CONSTEXPR_VARIABLE create_rebound_policy_t hpx::parallel::execution::create_rebound_policy = {}

struct create_rebound_policy_t

Public Functions

template<typename ExPolicy, typename Executor, typename Parameters>
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

template<>
using type = typename policy_type::template rebind::type
The type of the rebound execution policy.

namespace hpx

namespace execution

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

constexpr static_chunk_size()
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.

constexpr static_chunk_size(std::size_t chunk_size)
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.

namespace hpx

namespace parallel

namespace execution

Typedefs

template<typename Executor, typename T, typename ...Ts>
using executor_future_t = typename executor_future<Executor, T, Ts...>::type

template<typename Executor>
struct executor_context
Public Types

template<>
using type = std::decay_t<decltype(std::declval<Executor const&>().context())>

template< typename Executor>
struct executor_execution_category

Private Types

template< typename T>
using execution_category = typename T::execution_category

template<typename Executor>
struct executor_index

Public Types

template<>
using type = hpx::util::detected_or_t<hpx::execution::unsequenced_execution_tag, execution_category, Executor>

Private Types

template< typename T>
using index_type = typename T::index_type

template<typename Executor>
struct executor_parameters_type

Public Types

template<>
using type = hpx::util::detected_or_t<hpx::execution::static_chunk_size, parameters_type, Executor>

Private Types

template< typename T>
using parameters_type = typename T::parameters_type

template<typename Executor>
struct executor_shape
Public Types

template<>
using type = hpx::util::detected_or_t<std::size_t, shape_type, Executor>

Private Types

template<typename T>
using shape_type = typename T::shape_type

namespace traits

Typedefs

template<typename Executor>
using executor_context_t = typename executor_context<Executor>::type

template<typename Executor>
using executor_execution_category_t = typename executor_execution_category<Executor>::type

template<typename Executor>
using executor_shape_t = typename executor_shape<Executor>::type

template<typename Executor>
using executor_index_t = typename executor_index<Executor>::type

template<typename Executor, typename T, typename ...Ts>
using executor_future_t = typename executor_future<Executor, T, Ts...>::type

template<typename Executor>
using executor_parameters_type_t = typename executor_parameters_type<Executor>::type

Variables

template<typename T>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_post_member_v = has_post_member<T>::value

template<typename T>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_sync_execute_member_v = has_sync_execute_member<T>::value

template<typename T>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_async_execute_member_v = has_async_execute_member<T>::value

template<typename T>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_then_execute_member_v = has_then_execute_member<T>::value

template<typename T>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_bulk_sync_execute_member_v = has_bulk_sync_execute_member<T>::value

template<typename T>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_bulk_async_execute_member_v = has_bulk_async_execute_member<T>::value

template<typename T>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_bulk_then_execute_member_v = has_bulk_then_execute_member<T>::value

namespace hpx
Variables

template<typename T> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::is_execution_policy_v = is_execution_policy<T>::value

template<typename T> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::is_parallel_execution_policy_v = is_parallel_execution_policy<T>::value

template<typename T> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::is_sequenced_execution_policy_v = is_sequenced_execution_policy<T>::value

template<typename T> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::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>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy makes algorithms asynchronous

1. 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.

2. 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>.

3. 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>::type>
#include <is_execution_policy.hpp>

1. The type is_execution_policy can be used to detect execution policies for the purpose of excluding ambiguous overload resolution participation.

2. 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>.

3. 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>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy enables parallelization

1. 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.

2. 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>.

3. 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>::type>
#include <is_execution_policy.hpp> Extension: Detect whether given execution policy does not enable parallelization
1. 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.

2. 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>`.

3. The behavior of a program that adds specializations for `is_sequenced_execution_policy` is undefined.

```cpp
namespace hpx

namespace parallel

namespace traits

Functions

```cpp
std::size_t count_bits (bool value)
```

```cpp
execution_base

The contents of this module can be included with the header `hpx/modules/execution_base.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/execution_base.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace execution_base

struct agent_base

Public Functions

```cpp
virtual ~agent_base ()
virtual std::string description () const = 0
virtual context_base const &context () const = 0
virtual void yield (char const *desc) = 0
virtual void yield_k (std::size_t k, char const *desc) = 0
virtual void suspend (char const *desc) = 0
virtual void resume (char const *desc) = 0
virtual void abort (char const *desc) = 0
virtual void sleep_for (hpx::chrono::steady_duration const &sleep_duration, char const *desc) = 0
```
virtual void sleep_until(hpx::chrono::steady_time_point const &sleep_time, char const *desc) = 0

namespace hpx

namespace execution_base

class agent_ref

Public Functions

constexpr agent_ref()
constexpr agent_ref(agent_base *impl)
constexpr agent_ref(agent_ref const&)
constexpr agent_ref &operator=(agent_ref const&)
constexpr agent_ref &operator=(agent_ref&&)
constexpr agent_ref (agent_ref&&)

constexpr operator bool() const

void reset (agent_base *impl = nullptr)

void yield (char const *desc = "hpx::execution_base::agent::yield")

void yield_k (std::size_t k, char const *desc = "hpx::execution_base::agent::yield_k")

void suspend (char const *desc = "hpx::execution_base::agent::suspend")

void resume (char const *desc = "hpx::execution_base::agent::resume")

void abort (char const *desc = "hpx::execution_base::agent::abort")

template<typename Rep, typename Period>
void sleep_for (std::chrono::duration<Rep, Period> const &sleep_duration, char const *desc = "hpx::execution_base::agent::sleep_for")

template<typename Clock, typename Duration>
void sleep_until (std::chrono::time_point<Clock, Duration> const &sleep_time, char const *desc = "hpx::execution_base::agent::sleep_until")

agent_base &ref()

Private Functions

void sleep_for (hpx::chrono::steady_duration const &sleep_duration, char const *desc)

void sleep_until (hpx::chrono::steady_time_point const &sleep_time, char const *desc)
Private Members

agent_base *impl_

Friends

friend constexpr bool operator==(agent_ref const &lhs, agent_ref const &rhs)
friend constexpr bool operator!=(agent_ref const &lhs, agent_ref const &rhs)

namespace std
operator<<(std::ostream&, agent_ref const&)

namespace hpx

namespace execution

namespace experimental

template<typename ...Ts>
class any_sender

Public Types

template<template<typename...> class Tuple, template<typename...> class Variant> using value_types = Variant<Tuple<Ts...>>
template<typename...> class Variant> using error_types = Variant<std::exception_ptr>

Public Functions

any_sender()

template<typename Sender, typename = std::enable_if_t<std::is_same_v<std::decay_t<Sender>, any_sender>>>
any_sender(Sender &&sender)

template<typename Sender, typename = std::enable_if_t<std::is_same_v<std::decay_t<Sender>, any_sender>>>
any_sender &operator=(Sender &&&sender)

~any_sender()

any_sender(any_sender&&)

any_sender(any_sender const&)

any_sender &operator=(any_sender&&)

any_sender &operator=(any_sender const&)

2.8. API reference 879
**Public Static Attributes**

```cpp
constexpr bool sends_done = false
```

**Private Types**

```cpp
template<>
using base_type = detail::any_sender_base<Ts...>

template<typename Sender>
using impl_type = detail::any_sender_impl<Sender, Ts...>

template<>
using storage_type = hpx::detail::copyable_sbo_storage<base_type, 4 * sizeof(void*)>
```

**Private Members**

```cpp
storage_type storage = {}
```

**Friends**

```cpp
template<typename R>
detail::any_operation_state tag_invoke(hpx::execution::experimental::connect_t, any_sender &s, R &r)
```

```cpp
template<typename R>
detail::any_operation_state tag_invoke(hpx::execution::experimental::connect_t, any_sender &&s, R &&r)
```

```cpp
template<typename ...Ts>
class unique_any_sender
```

**Public Types**

```cpp
template<template<typename...> class Tuple, template<typename...> class Variant>
using value_types = Variant<Tuple<Ts...>>

template<template<typename...> class Variant>
using error_types = Variant<std::exception_ptr>
```

**Public Functions**

```cpp
unique_any_sender()
```

```cpp
template<typename Sender, typename = std::enable_if_t<!std::is_same_v<std::decay_t<Sender>, unique_any_sender>>
unique_any_sender(Sender &&sender)
```

```cpp
template<typename Sender, typename = std::enable_if_t<!std::is_same_v<std::decay_t<Sender>, unique_any_sender>>
unique_any_sender &operator= (Sender &&sender)
```

```cpp
~unique_any_sender()
```

```cpp
unique_any_sender(unique_any_sender&)
```
**unique_any_sender** *(unique_any_sender const&)*

unique_any_sender & operator= (unique_any_sender&&)

unique_any_sender & operator= (unique_any_sender const&)

### Public Static Attributes

`constexpr` bool `sends_done` = false

### Private Types

```cpp
template<>
using base_type = detail::unique_any_sender_base<Ts...>

template<typename Sender>
using impl_type = detail::unique_any_sender_impl<Sender, Ts...>

template<>
using storage_type = hpx::detail::movable_sbo_storage<base_type, 4 * sizeof(void*)>
```

### Private Members

```cpp
storage_type storage = {}
```

### Friends

```cpp
template<typename R>
detail::any_operation_state tag_invoke(hpx::execution::experimental::connect_t,
    unique_any_sender &&s, R &&r)
```

```cpp
namespace hpx

namespace execution

namespace experimental

Variables

```cpp
template<typename Scheduler>HPX_INLINE_CONSTEXPR_VARIABLE get_completion_scheduler_t<Scheduler> hpx::execution::experimental::get_completion_scheduler = {}
```

```cpp
namespace hpx

namespace execution_base

struct context_base
```
Public Functions

```cpp
virtual ~context_base()

virtual resource_base const & resource() const = 0
```

namespace hpx

```cpp
namespace execution

namespace experimental
```

Functions

```cpp
template<typename O>
void start(O && o)
```

start is a customization point object. The expression `hpx::execution::experimental::start(r)` is equivalent to:

- If the function selected does not signal the receiver `r`'s done channel, the program is ill-formed (no diagnostic required).
- Otherwise, `start(r)`, if that expression is valid, with overload resolution performed in a context that include the declaration `void start();`
- Otherwise, the expression is ill-formed.

The customization is implemented in terms of `hpx::functional::tag_invoke`.

Variables

```cpp
template<typename O> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::execution::experimental::is_operation_state_v = is_operation_state<O>::value
```

```cpp
template<typename O>
struct is_operation_state
```

An `operation_state` is an object representing the asynchronous operation that has been returned from calling `hpx::execution::experimental::connect` with a sender and a receiver. The only operation on an `operation_state` is:

- `hpx::execution::experimental::start` can be called exactly once. Once it has been invoked, the caller needs to ensure that the receiver's completion signaling operations strongly happen before the destructor of the state is called. The call to `hpx::execution::experimental::start` needs to happen strongly before the completion signaling operations.
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_done (R &&r)

set_done is a customization point object. The expression hpx::execution::set_done(r)
is equivalent to:
• r.set_done(), if that expression is valid. If the function selected does not signal the Re-
ceiver r’s done channel, the program is ill-formed (no diagnostic required).
• Otherwise, `set_done(r), if that expression is valid, with overload resolution performed in a
context that include the declaration void set_done();
• 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_done(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_done_t set_done

template<typename T, typename E = std::exception_ptr>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::execution::experimental::is_receiver_v = is_receiver<T, E>::value

template<typename T, typename... As>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::execution::experimental::is_receiver_of_v=is_receiver_of<T, As...>::value

template<typename T, typename... As>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::execution::experimental::is_nothrow_receiver_of_v=is_nothrow_receiver_of<T, As...>::value

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:
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 `hpx::execution::experimental::is_receiver_of`

```cpp
template<typename T, typename ...As>
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`
    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_done`

See `hpx::execution::traits::is_receiver`
```

namespace hpx

namespace util

Functions

```cpp
constexpr bool register_lock (void const*, util::register_lock_data* = nullptr)
constexpr bool unregister_lock (void const*)
constexpr void verify_no_locks ()
constexpr void force_error_on_lock ()
constexpr void enable_lock_detection ()
constexpr void disable_lock_detection ()
constexpr void trace_depth_lock_detection (std::size_t)
constexpr void ignore_lock (void const*)
constexpr void reset_ignored (void const*)
constexpr void ignore_all_locks ()
constexpr void reset_ignored_all ()
```
`std::unique_ptr<held_locks_data> get_held_locks_data()`

```cpp
constexpr void set_held_locks_data(std::unique_ptr<held_locks_data>&&) {
}
```

```cpp
struct ignore_all_while_checking

Public Functions

```cpp
constexpr ignore_all_while_checking() {
}
```

```cpp
template<
    typename Lock,
    typename Enable = void
>
struct ignore_while_checking

Public Functions

```cpp
constexpr ignore_while_checking(Lock const*) {
}
```

namespace hpx

namespace execution_base

```cpp
struct resource_base
#include <resource_base.hpp> TODO: implement, this is currently just a dummy.

Public Functions

```cpp
virtual ~resource_base() {
}
```

namespace hpx

namespace execution

namespace experimental

**Types**

```cpp
template<typename S, typename R>
using connect_result_t = typename hpx::util::invoke_result<
    connect_t, S, R>::type
```

**Functions**

```cpp
template<typename S, typename R>
void connect(S &&s, R &&r) {
connect is a customization point object. For some subexpression s and r, let S be the type such that `decltype((s))` is S and let R be the type such that `decltype((r))` is R. The result of the expression hpx::execution::experimental::connect(s, r) is then equivalent to:
```
• `s.connect(r)`, if that expression is valid and returns a type satisfying the `operation_state`
  {See `hpx::execution::experimental::traits::is_operation_state` and if `S` satisfies the `sender`
  concept.
• `s.connect(r)`, if that expression is valid and returns a type satisfying the `operation_state`
  {See `hpx::execution::experimental::traits::is_operation_state` and if `S` satisfies the `sender`
  concept. Overload resolution is performed in a context that include the declaration `void
  connect();`
• Otherwise, the expression is ill-formed.
The customization is implemented in terms of `hpx::functional::tag_invoke`.

**Variables**

```cpp
template<typename Sender> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::execution::experimental::is_sender_v = is_sender<Sender>::value
```

```cpp
hpx::execution::experimental::connect_t connect
hpx::execution::experimental::schedule_t schedule
```

```cpp
template<typename Scheduler> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::execution::experimental::is_scheduler_v = is_scheduler<Scheduler>::value
```

**Public Functions**

```cpp
void operator()()
```

```cpp
template<typename Sender>
struct is_sender
```

```cpp
#include <sender.hpp> The name schedule denotes a customization point object. For some subex-
pression `s`, let `S` be `decltype((s))`. The expression `schedule(s)` is expression-equivalent to:
```
```
* `s.schedule()`, *if* that expression is valid and its type models `sender`.
* Otherwise, `schedule(s)`, *if* that expression is valid and its type models `sender` with overload resolution performed in a context that includes the declaration

```cpp
void schedule();
```

and that does not include a declaration of `schedule`.

* Otherwise, `schedule(s)` is ill-formed.

The customization is implemented in terms of `hpx::functional::tag_invoke`. A sender
is a type that is describing an asynchronous operation. The operation itself might not have started
yet. In order to get the result of this asynchronous operation, a sender needs to be connected to a
receiver with the corresponding value, error and done channels:

- `hpx::execution::experimental::connect`

In addition, `hpx::execution::experimental::sender_traits` needs to be special-
ized in some form.

A sender’s destructor shall not block pending completion of submitted operations.
struct is_sender_to
#include <sender.hpp>

See is_sender

template<typename Sender>
struct sender_traits
#include <sender.hpp> sender_traits expose the different value and error types exposed
by a sender. This can be either specialized directly for user defined sender types or embedded
value_types, error_types and sends_done inside the sender type can be provided.

Subclassed by hpx::execution::experimental::sender_traits< Sender
& >, hpx::execution::experimental::sender_traits< Sender && >,
hx::execution::experimental::sender_traits< Sender const >,
hx::execution::experimental::sender_traits< Sender volatile >

template<(),
struct sender_traits<>()

Public Types

template<Any>
using __unspecialized = void

namespace hpx

namespace execution_base

namespace this_thread

Functions

hx::execution_base::agent_ref agent ()

void yield (char const *desc = "hx::execution_base::this_thread::yield")

void yield_k (std::size_t k, char const *desc = "hx::execution_base::this_thread::yield_k")

void suspend (char const *desc = "hx::execution_base::this_thread::suspend")

template<typename Rep, typename Period>
void sleep_for (std::chrono::duration<Rep, Period> const &sleep_duration, char const
*desc = "hx::execution_base::this_thread::sleep_for")

template<class Clock, class Duration>
void sleep_until (std::chrono::time_point<Clock, Duration> const &sleep_time, char const
*desc = "hx::execution_base::this_thread::sleep_for")

struct reset_agent
Public Functions

reset_agent (detail::agent_storage*, agent_base &impl)
reset_agent (agent_base &impl)
~reset_agent ()

Public Members

detail::agent_storage *storage_
agent_base *old_

namespace util

Functions

template<typename Predicate>
void yield_while (Predicate &&predicate, const char *thread_name = nullptr, bool allow_timed_suspension = true)

namespace hpx

namespace traits

Typedefs

template<typename T>
using is_one_way_executor_t = typename is_one_way_executor<T>::type

template<typename T>
using is_never_blocking_one_way_executor_t = typename is_never_blocking_one_way_executor<T>::type

template<typename T>
using is_bulk_one_way_executor_t = typename is_bulk_one_way_executor<T>::type

template<typename T>
using is_two_way_executor_t = typename is_two_way_executor<T>::type

template<typename T>
using is_bulk_two_way_executor_t = typename is_bulk_two_way_executor<T>::type

template<typename T>
using is_executor_any_t = typename is_executor_any<T>::type
Variables

```cpp
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_one_way_executor_v = is_one_way_executor<T>::value
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_never_blocking_one_way_executor_v = is_never_blocking_one_way_executor<T>::value
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_bulk_one_way_executor_v = is_bulk_one_way_executor<T>::value
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_two_way_executor_v = is_two_way_executor<T>::value
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_bulk_two_way_executor_v = is_bulk_two_way_executor<T>::value
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_executor_any_v = is_executor_any<T>::value
```

Public Types

```cpp
template<typename Executor>
struct extract_executor_parameters<Executor, typename hpx::util::always_void<typename Executor::executor_parameters_type>::type>
```

Public Types

```cpp
template<typename Parameters>
struct extract_has_variable_chunk_size<Parameters, typename hpx::util::always_void<typename Parameters::has_variable_chunk_size>::type>
```

namespace hpx

namespace parallel

namespace execution

Typedefs

```cpp
template<typename T>
using is_executor_parameters_t = typename is_executor_parameters<T>::type
```

Variables

```cpp
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::parallel::execution::is_executor_parameters_v = is_executor_parameters<T>::value
```
Public Types

```cpp
template<>
using type = sequential_executor_parameters
```

```cpp
template<typename Executor>
struct extract_executor_parameters<Executor, typename hpx::util::always_void<typename Executor::executor_parameters_type> type>
```

```cpp
template<typename Parameters, typename Enable = void>
struct extract_has_variable_chunk_size
```

```cpp
template<>
using type = std::false_type
```

```cpp
template<typename Parameters>
struct extract_has_variable_chunk_size<Parameters, typename hpx::util::always_void<typename Parameters::has_variable_chunk_size> type>
```

```cpp
template<>
using type = typename Parameters::has_variable_chunk_size
```

namespace traits

Typedefs

```cpp
template<typename T>
using is_executor_parameters_t = typename is_executor_parameters<T>::type
```

Variables

```cpp
template<typename T>HP_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_executor_parameters_v
```

exectors

The contents of this module can be included with the header `hpx/modules/executors.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/executors.hpp`, not the particular header in which the functionality you would like to use is defined. See `Public API` for a list of names that are part of the public HPX API.
namespace execution

namespace experimental

Functions

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>>>
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>>>
annotating_executor(Executor &&exec, std::string annotation)

namespace hpx

namespace parallel

namespace execution

Typedefs

using current_executor = parallel::execution::thread_pool_executor

namespace this_thread

Functions

parallel::execution::current_executor get_executor(error_code &ec = throws)

Returns a reference to the executor which was used to create the current thread.

Exceptions

• If: &ec != &throws, never throws, but will set ec to an appropriate value when an er-
ror occurs. Otherwise, this function will throw an hpx::exception with an error code of
hpx::yield_aborted if it is signaled with wait_aborted. If called outside of a HPX-thread, this
A function will throw an \texttt{hpx::exception} with an error code of \texttt{hpx::null_thread_id}. If this function is called while the thread-manager is not running, it will throw an \texttt{hpx::exception} with an error code of \texttt{hpx::invalid_status}.

\textbf{namespace threads}

\textbf{Functions}

\begin{verbatim}
parallel::execution::current_executor get_executor(thread_id_type const &id, error_code &ec = throws)
\end{verbatim}

Returns a reference to the executor which was used to create the given thread.

\textbf{Exceptions}

- If: \texttt{\&ec != \&throws}, never throws, but will set \texttt{ec} to an appropriate value when an error occurs. Otherwise, this function will throw an \texttt{hpx::exception} with an error code of \texttt{hpx::yield_aborted} if it is signaled with \texttt{wait_aborted}. If called outside of a HPX-thread, this function will throw an \texttt{hpx::exception} with an error code of \texttt{hpx::null_thread_id}. If this function is called while the thread-manager is not running, it will throw an \texttt{hpx::exception} with an error code of \texttt{hpx::invalid_status}.

\textbf{namespace hpx}

\textbf{namespace execution}

\textbf{Variables}

\begin{verbatim}
HPX_INLINE_CONSTEXPR_VARIABLE task_policy_tag hpx::execution::task = {}
\end{verbatim}

Default sequential execution policy object.

\begin{verbatim}
HPX_INLINE_CONSTEXPR_VARIABLE non_task_policy_tag hpx::execution::non_task = {}
\end{verbatim}

\begin{verbatim}
HPX_INLINE_CONSTEXPR_VARIABLE sequenced_policy hpx::execution::seq = {}
\end{verbatim}

Default sequential execution policy object.

\begin{verbatim}
constexpr parallel_policy par = {}
\end{verbatim}

Default parallel execution policy object.

\begin{verbatim}
HPX_INLINE_CONSTEXPR_VARIABLE parallel_unsequenced_policy hpx::execution::par_unseq = {}
\end{verbatim}

Default vector execution policy object.

\begin{verbatim}
HPX_INLINE_CONSTEXPR_VARIABLE unsequenced_policy hpx::execution::unseq = {}
\end{verbatim}

Default vector execution policy object.

\begin{verbatim}
struct parallel_policy
#include <execution_policy.hpp> The class \texttt{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.
\end{verbatim}
Public Types

using executor_type = parallel_executor
  The type of the executor associated with this execution policy.

using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type
  The type of the associated executor parameters object which is associated with this execution policy.

using execution_category = parallel_execution_tag
  The category of the execution agents created by this execution policy.

Public Functions

constexpr parallel_task_policy operator() (task_policy_tag) const
  Create a new parallel_policy referencing a chunk size.

Return  The new parallel_policy
Parameters
  • tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) operator() (non_task_policy_tag) const
  Create a new parallel_policy from itself

Return  The new parallel_policy
Parameters
  • tag: [in] Specify that the corresponding asynchronous execution policy should be used

template<typename Executor>
constexpr decltype(auto) on (Executor &&exec) const
  Create a new parallel_policy referencing an executor and a chunk size.

Return  The new parallel_policy
Parameters
  • exec: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with

template<typename ...Parameters>
constexpr decltype(auto) with (Parameters&... params) const
  Create a new parallel_policy from the given execution parameters

Note  Requires: is_executor_parameters<Parameters>::value is true
Return  The new parallel_policy
Template Parameters
  • Parameters: The type of the executor parameters to associate with this execution policy.
Parameters
  • params: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

executor_type &executor ()
  Return the associated executor object.
constexpr executor_type const &executor() const
    Return the associated executor object.

executor_parameters_type &parameters()
    Return the associated executor parameters object.

constexpr executor_parameters_type const &parameters() const
    Return the associated executor parameters object.

Private Functions

template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)

Private Members

executor_type exec_

executor_parameters_type params_

Friends

friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access

template<typename Executor_, typename Parameters_>
struct rebind
    #include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The
    execution category of Executor shall not be weaker than that of this execution policy.

Public Types

template<>
using type = parallel_policy_shim<Executor_, Parameters_>
    The type of the rebound execution policy.

template<typename Executor, typename Parameters>
struct parallel_policy_shim
    #include <execution_policy.hpp> The class parallel_policy_shim 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.
Public Types

template<>
using executor_type = std::decay_t<Executor>

The type of the executor associated with this execution policy.

template<>
using executor_parameters_type = std::decay_t<Parameters>

The type of the associated executor parameters object which is associated with this execution policy.

template<>
using execution_category = typename hpx::traits::executor_execution_category<executor_type>::type

The category of the execution agents created by this execution policy.

Public Functions

constexpr parallel_task_policy_shim<Executor, Parameters> operator(task_policy_tag) const

Create a new parallel_policy referencing a chunk size.

Return The new parallel_policy
Parameters
• tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) operator() (non_task_policy_tag) const

Create a new parallel_policy from itself

Return The new parallel_policy
Parameters
• tag: [in] Specify that the corresponding asynchronous execution policy should be used

template<typename Executor_>
constexpr decltype(auto) on (Executor_ &exec) const

Create a new parallel_policy from the given executor

Note Requires: is_executor<Executor>::value is true
Return The new parallel_policy
Template Parameters
• Executor: The type of the executor to associate with this execution policy.
Parameters
• exec: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

template<typename ...Parameters_>
constexpr decltype(auto) with (Parameters_&&... params) const

Create a new parallel_policy_shim from the given execution parameters

Note Requires: is_executor_parameters<Parameters>::value is true
Return The new parallel_policy_shim
Template Parameters
• Parameters: The type of the executor parameters to associate with this execution policy.
Parameters
• params: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

    executor_type &executor()
    Return the associated executor object.

def constexpr executor_type const &executor() const
    Return the associated executor object.

    executor_parameters_type &parameters()
    Return the associated executor parameters object.

def constexpr executor_parameters_type const &parameters() const
    Return the associated executor parameters object.

    template<typename Executor_, typename Parameters_>
    struct rebind
    
    #include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

    Public Types

    template<>
    template<>
    using type = parallel_policy_shim<Executor_, Parameters_>
    The type of the rebound execution policy.

    struct parallel_task_policy
    
    #include <execution_policy.hpp> 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.

    Public Types

    using executor_type = parallel_executor
    The type of the executor associated with this execution policy.

    using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type
    The type of the associated executor parameters object which is associated with this execution policy

    using execution_category = parallel::execution_tag
    The category of the execution agents created by this execution policy.
Public Functions

constexpr parallel_task_policy operator() (task_policy_tag) const
Create a new parallel_task_policy from itself

Return The new parallel_task_policy
Parameters
• tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) operator() (non_task_policy_tag) const
Create a new non task parallel policy from itself

Return The new non task parallel_policy
Parameters
• tag: [in] Specify that the corresponding execution policy should be used

template<typename Executor>
constexpr decltype(auto) on (Executor &&exec) const
Create a new parallel_task_policy from given executor

Note Requires: is_executor<Executor>::value is true
Return The new parallel_task_policy
Template Parameters
• Executor: The type of the executor to associate with this execution policy.
Parameters
• exec: [in] The executor to use for the execution of the parallel algorithm the returned
  execution policy is used with.

template<typename Parameters>
constexpr decltype(auto) with (Parameters &... params) const
Create a new parallel_policy_shim from the given execution parameters

Note Requires: all parameters are executor_parameters, different parameter types can’t be duplicated
Return The new parallel_policy_shim
Template Parameters
• Parameters: The type of the executor parameters to associate with this execution policy.
Parameters
• params: [in] The executor parameters to use for the execution of the parallel algorithm the
  returned execution policy is used with.

executor_type &executor ()
Return the associated executor object.

castexpr executor_type const &executor () const
Return the associated executor object.

executor_parameters_type &parameters ()
Return the associated executor parameters object.

castexpr executor_parameters_type const &parameters () const
Return the associated executor parameters object.
Private Functions

```cpp
template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)
```

Private Members

```cpp
executor_type exec_
executor_parameters_type params_
```

Friends

```cpp
friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access
```

template<typename Executor_, typename Parameters_>
```cpp
struct rebind
#include <execution_policy.hpp> // Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy
```

Public Types

```cpp
template<>
using type = parallel_task_policy_shim<Executor_, Parameters_>
```

```cpp
template<typename Executor, typename Parameters>
struct parallel_task_policy_shim
#include <execution_policy.hpp> // Extension: The class parallel_task_policy_shim is an execution policy type used as a unique type to disambiguate parallel algorithm overloading based on combining a underlying parallel_task_policy and an executor and indicate that a parallel algorithm's execution may be parallelized.
```

Public Types

```cpp
template<>
using executor_type = std::decay_t<Executor>
```

```cpp
template<>
using executor_parameters_type = std::decay_t<Parameters>
```

```cpp
template<>
using execution_category = typename hpx::traits::executor_execution_category<executor_type>::type
```

The category of the execution agents created by this execution policy.
Public Functions

**constexpr** parallel_task_policy_shim operator() (task_policy_tag) **const**
Create a new parallel_task_policy_shim from itself

Return The new **sequenced_task_policy**
Parameters
  • **tag** : [in] Specify that the corresponding asynchronous execution policy should be used

**constexpr** decltype(auto) operator() (non_task_policy_tag) **const**
Create a new non task parallel policy from itself

Return The new non task parallel_policy_shim
Parameters
  • **tag** : [in] Specify that the corresponding execution policy should be used

//template<typename Executor_>
**constexpr** decltype(auto) on (Executor_&& exec) **const**
Create a new parallel_task_policy from the given executor

Note Requires: is_executor<Executor>::value is true
Return The new parallel_task_policy
Template Parameters
  • **Executor**: The type of the executor to associate with this execution policy.
Parameters
  • **exec**: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

//template<typename ... Parameters_>
**constexpr** decltype(auto) with (Parameters_&&... params) **const**
Create a new parallel_policy_shim from the given execution parameters

Note Requires: all parameters are executor_parameters, different parameter types can’t be duplicated
Return The new parallel_policy_shim
Template Parameters
  • **Parameters**: The type of the executor parameters to associate with this execution policy.
Parameters
  • **params**: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

**executor_type & executor ()**
Return the associated executor object.

**constexpr** executor_type **const** & executor () **const**
Return the associated executor object.

**executor_parameters_type & parameters ()**
Return the associated executor parameters object.

**constexpr** executor_parameters_type **const** & parameters () **const**
Return the associated executor parameters object.

**template<typename Executor_, typename Parameters_>**
struct rebind
#include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy.

Public Types

template<>
template<>
using type = parallel_task_policy_shim<Executor_, Parameters_>
The type of the rebound execution policy.

struct parallel_unsequenced_policy
#include <execution_policy.hpp> 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.

Public Types

using executor_type = parallel_executor
The type of the executor associated with this execution policy.

using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type
The type of the associated executor parameters object which is associated with this execution policy.

using execution_category = parallel_execution_tag
The category of the execution agents created by this execution policy.

Public Functions

constexpr decltype(auto) operator() (non_task_policy_tag) const
Create a new non task policy from itself.

Return The non task parallel unsequenced policy.

executor_type &executor ()
Return the associated executor object.

constexpr executor_type const &executor () const
Return the associated executor object.

executor_parameters_type &parameters ()
Return the associated executor parameters object.

constexpr executor_parameters_type const &parameters () const
Return the associated executor parameters object.
Private Functions

template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)

Private Members

extutor_type exec_
executor_parameters_type params_

Friends

friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access

struct sequenced_policy
#include <execution_policy.hpp> 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.

Public Types

using executor_type = sequenced_executor
The type of the executor associated with this execution policy.
using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type
The type of the associated executor parameters object which is associated with this execution
policy
using execution_category = sequenced_execution_tag
The category of the execution agents created by this execution policy.

Public Functions

constexpr sequenced_task_policy operator() (task_policy_tag) const
Create a new sequenced_task_policy.

Return The new sequenced_task_policy
Parameters
• tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) operator() (non_task_policy_tag) const
Create a new sequenced_policy from itself.

Return The new sequenced_policy

template<typename Executor>
constexpr decltype(auto) on (Executor &&exec) const
Create a new sequenced_policy from the given executor
Note Requires: is_executor<Executor>::value is true
Return The new sequenced_policy
Template Parameters
  - Executor: The type of the executor to associate with this execution policy.
Parameters
  - exec: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

```cpp
template<typename ...Parameters>
constexpr decltype(auto) with (Parameters&&... params) const
  Create a new sequenced_policy from the given execution parameters
```

Note Requires: all parameters are executor_parameters, different parameter types can’t be duplicated
Return The new sequenced_policy
Template Parameters
  - Parameters: The type of the executor parameters to associate with this execution policy.
Parameters
  - params: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

```cpp
executor_type &executor ()
  Return the associated executor object.
```

```cpp
constexpr executor_type const &executor () const
  Return the associated executor object.
```

```cpp
executor_parameters_type &parameters ()
  Return the associated executor parameters object.
```

```cpp
constexpr executor_parameters_type const &parameters () const
  Return the associated executor parameters object.
```

Private Functions

```cpp
template<typename Archive>
constexpr void serialize (Archive&, const unsigned int)
```

Private Members

```cpp
executor_type exec_
```

```cpp
executor_parameters_type params_
```
Friends

friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access

template<typename Executor_, typename Parameters_>
struct rebind
#include <execution_policy.hpp>
Rebind the type of executor used by this execution policy. The
execution category of Executor shall not be weaker than that of this execution policy

Public Types

template<>
using type = sequenced_policy_shim<Executor_, Parameters_>
The type of the rebound execution policy.

template<typename Executor, typename Parameters>
struct sequenced_policy_shim
#include <execution_policy.hpp>
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.

Public Types

template<>
using executor_type = std::decay_t<Executor>
The type of the executor associated with this execution policy.

template<>
using executor_parameters_type = std::decay_t<Parameters>
The type of the associated executor parameters object which is associated with this execution
policy.

template<>
using execution_category = typename hpx::traits::executor_execution_category<executor_type>::type
The category of the execution agents created by this execution policy.

Public Functions

constexpr sequenced_task_policy_shim<Executor, Parameters> operator() (task_policy_tag) const
Create a new sequenced_task_policy.

Return The new sequenced_task_policy_shim
Parameters
• tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) operator() (non_task_policy_tag) const
Create a new sequenced_policy from itself.

Return The new sequenced_policy_shim
template<typename Executor_>
constexpr decltype(auto) on (Executor_&& exec) const
Create a new sequenced_policy from the given executor

Note Requires: is_executor<Executor>::value is true
Return The new sequenced_policy

Template Parameters
• Executor: The type of the executor to associate with this execution policy.

Parameters
• exec: [in] The executor to use for the execution of the parallel algorithm the returned
  execution policy is used with.

template<typename ...Parameters_>
constexpr decltype(auto) with (Parameters_&&... params) const
Create a new sequenced_policy_shim from the given execution parameters

Note Requires: all parameters are executor_parameters, different parameter types can’t be duplicated
Return The new sequenced_policy_shim

Template Parameters
• Parameters: The type of the executor parameters to associate with this execution policy.

Parameters
• params: [in] The executor parameters to use for the execution of the parallel algorithm the
  returned execution policy is used with.

executor_type &executor()
Return the associated executor object.

constexpr executor_type const &executor() const
Return the associated executor object.

executor_parameters_type &parameters()
Return the associated executor parameters object.

constexpr executor_parameters_type const &parameters() const
Return the associated executor parameters object.

template<typename Executor_, typename Parameters_>
struct rebind
#include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The
execution category of Executor shall not be weaker than that of this execution policy

Public Types

template<>
template<>
using type = sequenced_policy_shim<Executor_, Parameters_>
The type of the rebound execution policy.

struct sequenced_task_policy
#include <execution_policy.hpp> 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 \textit{sequenced_policy}.

**Public Types**

\texttt{using} \texttt{executor\_type} = \texttt{sequenced\_executor}  

The type of the executor associated with this execution policy.

\texttt{using} \texttt{executor\_parameters\_type} = \texttt{parallel::execution::extract\_executor\_parameters<executor\_type>::type}  

The type of the associated executor parameters object which is associated with this execution policy.

\texttt{using} \texttt{execution\_category} = \texttt{sequenced\_execution\_tag}  

The category of the execution agents created by this execution policy.

**Public Functions**

\texttt{constexpr sequenced\_task\_policy operator() (task\_policy\_tag) \ const}  

Create a new \texttt{sequenced\_task\_policy} from itself

\textbf{Return} The new \texttt{sequenced\_task\_policy}  

\textbf{Parameters}  

- \texttt{tag}: [in] Specify that the corresponding asynchronous execution policy should be used

\texttt{constexpr decltype(auto) operator() (non\_task\_policy\_tag) \ const}  

Create a corresponding non task policy for this task policy

\textbf{Return} The non task sequential policy

\texttt{template<typename Executor> constexpr decltype(auto) on (Executor \&\& exec) \ const}  

Create a new \texttt{sequenced\_task\_policy} from the given executor

\textbf{Note} Requires: \texttt{is\_executor<Executor>::value} is true

\textbf{Return} The new \texttt{sequenced\_task\_policy}  

\textbf{Template Parameters}  

- \texttt{Executor}: The type of the executor to associate with this execution policy.

\textbf{Parameters}  

- \texttt{exec}: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

\texttt{template<typename... Parameters> constexpr decltype(auto) with (Parameters\&\&... params) \ const}  

Create a new \texttt{sequenced\_task\_policy} from the given execution parameters

\textbf{Note} Requires: all parameters are \texttt{executor\_parameters}, different parameter types can’t be duplicated

\textbf{Return} The new \texttt{sequenced\_task\_policy}  

\textbf{Template Parameters}  

- \texttt{Parameters}: The type of the executor parameters to associate with this execution policy.

\textbf{Parameters}
• params: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.

`executor_type &executor()`
Return the associated executor object.

`constexpr executor_type const &executor() const`
Return the associated executor object.

`executor_parameters_type &parameters()`
Return the associated executor parameters object.

`constexpr executor_parameters_type const &parameters() const`
Return the associated executor parameters object.

Private Functions

`template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)`

Private Members

`executor_type exec_`
`executor_parameters_type params_`

Friends

`friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t`
`friend hpx::execution::hpx::serialization::access`

`template<typename Executor_, typename Parameters_>
struct rebind`
#include <execution_policy.hpp> Rebind the type of executor used by this execution policy. The execution category of Executor shall not be weaker than that of this execution policy

Public Types

`template<>`
using type = sequenced_task_policy_shim<Executor_, Parameters_>
The type of the rebound execution policy.

`template<typename Executor, typename Parameters>
struct sequenced_task_policy_shim`
#include <execution_policy.hpp> Extension: The class sequenced_task_policy_shim is an execution policy type used as a unique type to disambiguate parallel algorithm overloading based on combining a underlying sequenced_task_policy and an executor 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.
Public Types

template<>
using executor_type = std::decay_t<Executor>

The type of the executor associated with this execution policy.

template<>
using executor_parameters_type = std::decay_t<Parameters>

The type of the associated executor parameters object which is associated with this execution policy.

typedef hpx::traits::executor_execution_category<executor_type>::type execution_category
The category of the execution agents created by this execution policy.

Public Functions

constexpr sequenced_task_policy_shim const & operator() (task_policy_tag) const
Create a new sequenced_task_policy from itself

  Return  The new sequenced_task_policy
  Parameters
  •  tag: [in] Specify that the corresponding asynchronous execution policy should be used

constexpr decltype(auto) operator() (non_task_policy_tag) const
Create a corresponding non task policy for this task policy

  Return  The non task sequential shim policy

template<typename Executor_>
constexpr decltype(auto) on (Executor_ && exec) const
Create a new sequenced_task_policy from the given executor

  Note  Requires: is_executor<Executor>::value is true
  Return  The new sequenced_task_policy
  Template Parameters
  •  Executor: The type of the executor to associate with this execution policy.
  Parameters
  •  exec: [in] The executor to use for the execution of the parallel algorithm the returned execution policy is used with.

template<typename ...Parameters_>
constexpr decltype(auto) with (Parameters_&&... params) const
Create a new sequenced_task_policy_shim from the given execution parameters

  Note  Requires: all parameters are executor_parameters, different parameter types can’t be duplicated
  Return  The new sequenced_task_policy_shim
  Template Parameters
  •  Parameters: The type of the executor parameters to associate with this execution policy.
  Parameters
  •  params: [in] The executor parameters to use for the execution of the parallel algorithm the returned execution policy is used with.
executor_type &executor()  
    Return the associated executor object.

constexpr executor_type const &executor() const  
    Return the associated executor object.

executor_parameters_type &parameters()  
    Return the associated executor parameters object.

constexpr executor_parameters_type const &parameters() const  
    Return the associated executor parameters object.

template<typename Executor_, typename Parameters_>  
    struct rebind  
        #include <execution_policy.hpp>  
        Rebind the type of executor used by this execution policy. The 
        execution category of Executor shall not be weaker than that of this execution policy

Public Types

template<>  
    template<>  
        using type = sequenced_task_policy_shim<Executor_, Parameters_>  
        The type of the rebound execution policy.

struct unsequenced_policy  
    #include <execution_policy.hpp>  
    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.

Public Types

using executor_type = sequenced_executor  
    The type of the executor associated with this execution policy.

using executor_parameters_type = parallel::execution::extract_executor_parameters<executor_type>::type  
    The type of the associated executor parameters object which is associated with this execution 
    policy

using execution_category = sequenced_execution_tag  
    The category of the execution agents created by this execution policy.

Public Functions

cconstexpr decltype(auto) operator() (non_task_policy_tag) const  
    Create a new non task policy from itself

Return  
    The non task unsequenced policy

executor_type &executor()  
    Return the associated executor object.

constexpr executor_type const &executor() const  
    Return the associated executor object.
executor_parameters_type& parameters()
Return the associated executor parameters object.

constexpr executor_parameters_type const &parameters() const
Return the associated executor parameters object.

Private Functions

template<typename Archive>
constexpr void serialize(Archive&, const unsigned int)

Private Members

executor_type exec_
executor_parameters_type params_

Friends

friend hpx::execution::hpx::parallel::execution::create_rebound_policy_t
friend hpx::execution::hpx::serialization::access

namespace parallel

namespace execution

Typedefs

typedef hpx::execution::sequenced_executor instead

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)
namespace hpx

namespace execution

namespace experimental

class fork_join_executor

#include <fork_join_executor.hpp> An executor with fork-join (blocking) semantics.

The *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.

**Public Types**

**enum loop_schedule**

Type of loop schedule for use with the *fork_join_executor*. loop_schedule::static_ implies no work-stealing; loop_schedule::dynamic allows stealing when a worker has finished its local work.

*Values:*

- static_
- dynamic

**Public Functions**

**fork_join_executor** *(threads::thread_priority priority = threads::thread_priority::high, threads::thread_stacksize stacksize = threads::thread_stacksize::small_, loop_schedule schedule = loop_schedule::static_, std::chrono::nanoseconds yield_delay = std::chrono::milliseconds(1))*

Construct a *fork_join_executor*.

**Parameters**

- **priority**: The priority of the worker threads.
- **stacksize**: The stacksize of the worker threads.
- **schedule**: The loop schedule of the parallel regions.
- **yield_delay**: The time after which the executor yields to other work if it hasn’t received any new work for bulk execution.
Defines

GUIDED_POOL_EXECUTOR_DEBUG

namespace hpx

Functions

static hpx::debug::enable_print<GUIDED_POOL_EXECUTOR_DEBUG> hpx::gpx_deb("GP_EXEC")

namespace parallel

namespace execution

template<typename Hint>
struct executor_execution_category<guided_pool_executor<Hint>>

Public Types

typedef hpx::execution::parallel_execution_tag type

template<typename Hint>
struct executor_execution_category<guided_pool_executor_shim<Hint>>

Public Types

typedef hpx::execution::parallel_execution_tag type

template<typename Tag>
struct guided_pool_executor<pool_numa_hint<Tag>>

Public Functions

guided_pool_executor (threads::thread_pool_base *pool, bool hp_sync = false)
guided_pool_executor (threads::thread_pool_base *pool, threads::thread_stacksize stacksize, bool hp_sync = false)
guided_pool_executor (threads::thread_pool_base *pool, threads::thread_priority priority, threads::thread_stacksize stacksize = threads::thread_stacksize::default_, bool hp_sync = false)

template<typename F, typename ...Ts>
future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute (F &&f, Ts &&...ts)

template<typename F, typename Future, typename ...Ts, typename = std::enable_if_t<hpx::traits::is_future<Future>>
auto then_execute (F &&f, Future &&&predecessor, Ts &&... ts)

template<typename F, template<typename> class OuterFuture, typename ...InnerFutures, typename ...Ts>
auto then_execute (F &&f, OuterFuture<hpx::tuple<InnerFutures...>> &&predecessor, Ts&&... ts)

template<typename F, typename ...InnerFutures, typename = std::enable_if_t<hpx::traits::is_future_tuple<InnerFutures...>::value>>
auto async_execute (F &&f, hpx::tuple<InnerFutures...> &&predecessor)

Private Members

threads::thread_pool_base *pool_
threads::thread_priority priority_
threads::thread_stacksize stacksize_
pool_numa_hint<Tag> hint_
bool hp_sync_

Friends

friend hpx::parallel::execution::guided_pool_executor_shim

template<typename H>
struct guided_pool_executor_shim

Public Functions

guided_pool_executor_shim (bool guided, threads::thread_pool_base *pool, bool hp_sync = false)

guided_pool_executor_shim (bool guided, threads::thread_pool_base *pool, threads::thread_stacksize stacksize, bool hp_sync = false)

guided_pool_executor_shim (bool guided, threads::thread_pool_base *pool, threads::thread_priority priority, threads::thread_stacksize stacksize = threads::thread_stacksize::default_, bool hp_sync = false)

template<typename F, typename ...Ts>
future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute (F &&f, Ts&&... ts)

template<typename F, typename Future, typename ...Ts, typename = std::enable_if_t<hpx::traits::is_future<Future>::value>>
auto then_execute (F &&f, Future &&&predecessor, Ts&&... ts)
Public Members

bool guided_

guided_pool_executor<H> guided_exec_

namespace hpx

namespace execution

namespace experimental

Typedefs

using print_on = hpx::debug::enable_print<false>

Functions

static constexpr print_on hpx::execution::experimental::lim_debug("LIMEXEC")

template<typename BaseExecutor>
struct limiting_executor

Public Types

template<>
using execution_category = typename BaseExecutor::execution_category

template<>
using executor_parameters_type = typename BaseExecutor::executor_parameters_type

Public Functions

limiting_executor (BaseExecutor &ex, std::size_t lower, std::size_t upper, bool block_on_destruction = true)

limiting_executor (std::size_t lower, std::size_t upper, bool block_on_destruction = true)

~limiting_executor ()

limiting_executor const &context () const

template<typename F, typename ...Ts>
decltype(auto) sync_execute (F &&f, Ts&&... ts) const

template<typename F, typename ...Ts>
decltype(auto) async_execute (F &&f, Ts&&... ts)

template<typename F, typename Future, typename ...Ts>
decltype(auto) then_execute (F &&f, Future &&&predecessor, Ts&&... ts)

template<typename F, typename ...Ts>
void `post` \((F \&\& f, Ts\&\&... ts)\)

template<typename \(F\), typename \(S\), typename ...\(Ts\)>
dcltype(auto) `bulk_async_execute` \((F \&\& f, S const &shape, Ts\&\&... ts)\)

template<typename \(F\), typename \(S\), typename \(Future\), typename ...\(Ts\)>
dcltype(auto) `bulk_then_execute` \((F \&\& f, S const &shape, Future \&\& predecessor, Ts\&\&... ts)\)

void `wait` ()

void `wait_all` ()

void `set_threshold` \((std::size_t lower, std::size_t upper)\)

**Private Functions**

void `count_up` ()

void `count_down` () const

void `set_and_wait` \((std::size_t lower, std::size_t upper)\)

**Private Members**

BaseExecutor `executor_`

`std::atomic<std::size_t> count_`

`std::size_t lower_threshold_`

`std::size_t upper_threshold_`

bool `block_`

struct `on_exit`

**Public Functions**

template<>
`on_exit` \((limiting_executor const &this_e)\)

template<>
`~on_exit` ()

**Public Members**

template<>
`limiting_executor const &executor_`

template<typename \(F\), typename \(B = BaseExecutor\), typename \(Enable = void\)>`struct throttling_wrapper`
Public Functions

```cpp
template<>
throttling wrapper (limiting_executor &lim, BaseExecutor const&, F &f)
```

```cpp
template<typename ...Ts>
dcltype(auto) operator() (Ts&&... ts)
```

```cpp
template<>
bool exceeds_upper ()
```

```cpp
template<>
bool exceeds_lower ()
```

Public Members

```cpp
template<>
limiting_executor &limiting_
```

```cpp
template<>
F f_
```

namespace hpx

namespace execution

Typedefs

```cpp
using parallel_executor = parallel_policy_executor<hpx::launch>
```

```cpp
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

```cpp
template<>
using execution_category = parallel_execution_tag
```

Associate the parallel_execution_tag executor tag type as a default with this executor.

```cpp
template<>
using executor_parameters_type = static_chunk_size
```

Associate the static_chunk_size executor parameters type as a default with this executor.
Public Functions

```cpp
constexpr parallel_policy_executor (threads::thread_priority priority,
                                      threads::thread_stacksize stacksize = threads::thread_stacksize::default_,
                                      threads::thread_schedule_hint sched-ulehint = {}, Policy l = parallel::execution::detail::get_default_policy<Policy>::call(),
                                      std::size_t hierarchical_threshold = hierarchical_threshold_default_)
```
Create a new parallel executor.

```cpp
constexpr parallel_policy_executor (threads::thread_stacksize stacksize,
                                      threads::thread_schedule_hint sched-ulehint = {}, Policy l = parallel::execution::detail::get_default_policy<Policy>::call())
```

```cpp
constexpr parallel_policy_executor (threads::thread_schedule_hint schedulehint, Policy l = parallel::execution::detail::get_default_policy<Policy>::call())
```

```cpp
constexpr parallel_policy_executor (Policy l = parallel::execution::detail::get_default_policy<Policy>::call())
```

```cpp
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 sched-ulehint = {}, Policy l = parallel::execution::detail::get_default_policy<Policy>::call(),
                                      std::size_t hierarchical_threshold = hierarchical_threshold_default_)
```

Friends

```cpp
friend constexpr parallel_policy_executor tag_invoke (hpx::execution::experimental::with_hint_t,
                                                      parallel_policy_executor const &exec,
                                                      hpx::threads::thread_schedule_hint hint)
```

```cpp
friend constexpr hpx::threads::thread_schedule_hint tag_invoke (hpx::execution::experimental::get_hint_t,
                                                                   parallel_policy_executor const &exec)
```

```cpp
friend constexpr parallel_policy_executor tag_invoke (hpx::execution::experimental::with_priority_t,
                                                      parallel_policy_executor const &exec,
                                                      hpx::threads::thread_priority priority)
```

```cpp
friend constexpr hpx::threads::thread_priority tag_invoke (hpx::execution::experimental::get_priority_t,
                                                           parallel_policy_executor const &exec)
```

916 Chapter 2. What’s so special about HPX?
friend constexpr parallel_policy_executor tag_invoke (hpx::execution::experimental::with_annotation_t, parallel_policy_executor const &exec, char const *annotation)

parallel_policy_executor tag_invoke (hpx::execution::experimental::with_annotation_t, parallel_policy_executor const &exec, std::string annotation)

friend constexpr char const *tag_invoke (hpx::execution::experimental::get_annotation_t, parallel_policy_executor const &exec)

template<>
struct parallel_policy_executor_aggregated

Public Types

template<>
using execution_category = hpx::execution::parallel_execution_tag
Associate the parallel_execution_tag executor tag type as a default with this executor.

template<>
using executor_parameters_type = hpx::execution::static_chunk_size
Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

cast expr parallel_policy_executor_aggregated (hpx::launch l = hpx::launch::async_policy{}, std::size_t spread = 4, std::size_t tasks = std::size_t(-1))

Create a new parallel executor.

template<typename F, typename S, typename ...Ts>
std::vector<hpx::future<void>> bulk_async_execute (F &&f, S const &shape, Ts&&... ts)

namespace hpx

namespace parallel

namespace execution

Typedefs

using parallel_executor_aggregated = parallel_policy_executor_aggregated<hpx::launch::async_policy>

template<typename Policy = hpx::launch::async_policy>
struct parallel_policy_executor_aggregated
#include <parallel_executor_aggregated.hpp> A parallel_executor_aggregated creates groups of parallel execution agents that 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

```cpp
template<>
using execution_category = hpx::execution::parallel_execution_tag
    // Associate the parallel_execution_tag executor tag type as a default with this executor.

template<>
using executor_parameters_type = hpx::execution::static_chunk_size
    // Associate the static_chunk_size executor parameters type as a default with this executor.
```

Public Functions

```cpp
constexpr parallel_policy_executor_aggregated
    (std::size_t spread = 4, std::size_t tasks = std::size_t(-1))
    // Create a new parallel executor.

template<typename F, typename S, typename ...Ts>
std::vector<hpx::future<void>> bulk_async_execute
    (F &&f, S const &shape, Ts&&... ts) const
```

```cpp
namespace hpx

namespace parallel

namespace execution
```

Chapter 2. What’s so special about HPX?
class restricted_thread_pool_executor

Public Types

typedef hpx::execution::parallel_execution_tag execution_category
Associate the parallel_execution_tag executor tag type as a default with this executor.

typedef hpx::execution::static_chunk_size executor_parameters_type
Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

restricted_thread_pool_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 schedule_hint = {}, std::size_t hierarchical_threshold = hierarchical_threshold_default_)
Create a new parallel executor.

restricted_thread_pool_executor (restricted_thread_pool_executor &other) const

Private Members

threads::thread_pool_base *pool_ = nullptr
threads::thread_priority priority_ = threads::thread_priority::default_
threads::thread_stacksize stacksize_ = threads::thread_stacksize::default_
threads::thread_schedule_hint schedule_hint_ = {}
std::size_t hierarchical_threshold_ = hierarchical_threshold_default_
std::size_t first_thread_
std::size_t num_threads_
std::atomic<std::size_t> os_thread_

Private Static Attributes

constexpr std::size_t hierarchical_threshold_default_ = 6

namespace hpx

namespace execution

namespace experimental

2.8. API reference 919
Functions

template<typename BaseScheduler>
scheduler_executor (BaseScheduler && sched)

template<typename BaseScheduler>
struct scheduler_executor

Public Functions

constexpr scheduler_executor ()

template<typename Scheduler, typename Enable = std::enable_if_t<brp::execution::experimental::is_scheduler_v<Scheduler>>>
constexpr scheduler_executor (Scheduler && sched)

constexpr scheduler_executor (scheduler_executor&&)

constexpr scheduler_executor & operator= (scheduler_executor&&)

constexpr scheduler_executor (scheduler_executor const &)

constexpr scheduler_executor & operator= (scheduler_executor const &)

namespace hpx

namespace execution

struct sequenced_executor

#include <sequenced_executor.hpp> A sequenced_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 hpx

namespace parallel

namespace execution

Typedefs

using thread_pool_executor = hpx::execution::parallel_executor

namespace hpx

namespace execution

namespace experimental

struct thread_pool_scheduler
Public Functions

```cpp
constexpr thread_pool_scheduler()

thread_pool_scheduler (hpx::threads::thread_pool_base *pool)
```

namespace hpx

    namespace execution

        namespace experimental

            Functions

```cpp
template<typename Sender, typename Shape, typename F>
constexpr auto tag_invoke (bulk_t, thread_pool_scheduler scheduler, Sender &&sender, Shape &&shape, F &&f)
```

filesystem

The contents of this module can be included with the header hpx/modules/filesystem.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/filesystem.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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

```cpp
path initial_path ()

std::string basename (path const &p)

path canonical (path const &p, path const &base)

path canonical (path const &p, path const &base, std::error_code &ec)
```
format

The contents of this module can be included with the header hpx/modules/format.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/format.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

DECL_TYPE_SPECIFIER (Type, Spec)

namespace hpx

namespace util

Functions

template<typename ...Args>
std::string format (boost::string_ref format_str, Args const&... args)

template<typename ...Args>
std::ostream &format_to (std::ostream &os, boost::string_ref format_str, Args const&... args)

template<typename Range>
detail::format_join<Range> format_join (Range const &range, boost::string_ref delimiter)

namespace hpx

namespace util

class bad_lexical_cast : public bad_cast

Public Functions

bad_lexical_cast ()

const char *what () const

virtual ~bad_lexical_cast ()

bad_lexical_cast (std::type_info const &source_type_arg, std::type_info const &target_type_arg)

std::type_info const &source_type () const

std::type_info const &target_type () const
Private Members

```cpp
std::type_info const *source
std::type_info const *target
```

namespace hpx

namespace util

Functions

```cpp
template<typename T, typename Char>
T from_string(std::basic_string<Char> const &v)

template<typename T, typename U, typename Char>
T from_string(std::basic_string<Char> const &v, U &&default_value)

template<typename T>
T from_string(std::string const &v)

template<typename T, typename U>
T from_string(std::string const &v, U &&default_value)
```

namespace hpx

namespace util

Functions

```cpp
template<typename T>
std::string to_string(T const &v)
```

**functional**

The contents of this module can be included with the header `hpx/modules/functional.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/functional.hpp`, not the particular header in which the functionality you would like to use is defined. See **Public API** for a list of names that are part of the public HPX API.

namespace hpx

namespace serialization
Functions

```cpp
template<typename Archive, typename F, typename ...Ts>
void serialize(Archive &ar, ::hpx::util::detail::bound<F, Ts...> &bound, unsigned int const version = 0)
```

```cpp
namespace util {

Functions

```cpp
namespace util {

namespace serialization {

Functions

```cpp
namespace util {

Variables

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<1> hpx::util::placeholders::_1 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<2> hpx::util::placeholders::_2 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<3> hpx::util::placeholders::_3 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<4> hpx::util::placeholders::_4 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<5> hpx::util::placeholders::_5 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<6> hpx::util::placeholders::_6 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<7> hpx::util::placeholders::_7 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<8> hpx::util::placeholders::_8 = {}
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE detail::placeholder<9> hpx::util::placeholders::_9 = {}
```

namespace hpx

namespace serialization {

Functions

```cpp
namespace util {

```
Functions

template<typename F, typename ...Ts>
constexpr detail::bound_back<
    typename std::decay<F>::type,
    typename util::make_index_pack<
        sizeof...(Ts)
    >::type,
    typename util::decay_unwrap<Ts>::type...>
bind_back(F&& f, Ts&&... vs)

template<typename F>
constexpr std::decay<F>::type
bind_back(F&& f)

namespace hpx

namespace serialization

Functions

template<typename Archive, typename F, typename ...Ts>
void serialize(Archive&, ::hpx::util::detail::bound_front<F, Ts...> &bound, unsigned int
    const version = 0)

namespace util

Functions

template<typename F, typename ...Ts>
constexpr detail::bound_front<
    typename std::decay<F>::type,
    typename util::make_index_pack<
        sizeof...(Ts)
    >::type,
    typename util::decay_unwrap<Ts>::type...>
bind_front(F&& f, Ts&&... vs)

template<typename F>
constexpr std::decay<F>::type
bind_front(F&& f)

namespace hpx

namespace serialization

Functions

template<typename Archive, typename F, typename ...Ts>
void serialize(Archive&, ::hpx::util::detail::deferred<F, Ts...> &d, unsigned int const version = 0)

namespace util
Functions

template<typename F, typename ...Ts>
detail::deferred<
std::decay_t<F>,
util::make_index_pack_t<sizeof...(Ts)>,
util::decay_unwrap_t<Ts>...>
defered_call


template<typename F>
std::decay_t<F> deferred_call (F &&f)

Defines

HPX_UTIL_REGISTER_FUNCTION_DECLARATION (Sig, F, Name)
HPX_UTIL_REGISTER_FUNCTION (Sig, F, Name)

namespace hpx

namespace util

Typedefs

template<typename Sig>
using function_nonser = function<Sig, false>

template<typename R, typename ...Ts, bool Serializable>
class function<R (Ts...) , Serializable> : public detail::basic_function<R
Ts..., true, Serializable>

Public Types

template<>
using result_type = R

Public Functions

constexpr function (std::nullptr_t = nullptr)

function (function const&)

function (function&&)

function &operator= (function const&)

function &operator= (function&&)

template<typename F, typename FD = typename std::decay<F>::type, typename Enable1 = typename std::enable_if<!std::is_same<
FD, function>::value>::type, typename Enable2 = typename std::enable_if<is_invocable_r_v<R,
FD&, Ts...>>::type>

function (F &&f)


template<typename F, typename FD = typename std::decay<F>::type, typename Enable1 = typename std::enable_if<!std::is_same<
FD, function>::value>::type, typename Enable2 = typename std::enable_if<is_invocable_r_v<R,
FD&, Ts...>>::type>

function &operator= (F &&f)
Private Types

template<>
using base_type = detail::basic_function<R (Ts...), true, Serializable>

namespace hpx

namespace util

template<typename R, typename ...Ts>
class function_ref<R (Ts...)>

Public Functions

template<typename F, typename FD = typename std::decay<F>::type, typename Enable = typename std::enable_if<!std::is_same<FD, function_ref>::value && is_invocable_r_v<R, F&, Ts...>>::type>
function_ref(F&& f)

function_ref(function_ref const& other)

function_ref& operator=(F&& f)

function_ref& operator=(function_ref const& other)

template<typename F, typename T = typename std::remove_reference<F>::type, typename Enable = typename std::enable_if<!std::is_pointer<T>::value>::type>
void assign(F&& f)

void assign(std::reference_wrapper<T> f_ref)

void assign(T*f_ptr)

void swap(function_ref &f)

R operator()(Ts... vs) const

std::size_t get_function_address() const

cchar const*get_function_annotation() const

util::itt::string_handle get_function_annotation_itt() const

Protected Attributes

template<>
R (*vptr)(void*, Ts&&...)

void *object
Private Types

template<>
using VTable = detail::function_ref_vtable<R (Ts...)>

Private Static Functions

template<typename T>
static VTable const *get_vtable()

Defines

HPX_INVOKE_R (R, F, ...)

amespace hpx

namespace util

Functions

template<typename F, typename ...Ts>
constexpr util::invoke_result<F, Ts...>::type invoke (F &&f, Ts&&... vs)

Invokes the given callable object f with the content of the argument pack vs

Return  The result of the callable object when it’s called with the given argument types.
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
Exceptions
• std::exception: like objects thrown by call to object f with the argument types vs.

template<typename R, typename F, typename ...Ts>
constexpr R invoke_r (F &&f, Ts&&... vs)

Template Parameters
• R: The result type of the function when it’s called with the content of the given argument types
  vs.

namespace functional

struct invoke
Public Functions

```cpp
template<typename F, typename ...Ts>
constexpr util::invoke_result<F, Ts...>::type operator() (F &&f, Ts&&... vs) const
```

```cpp
namespace hpx

namespace util

Functions

```cpp
template<typename F, typename Tuple>
constexpr detail::invoke_fused_result<F, Tuple>::type invoke_fused (F &&f, Tuple &&t)
```

Invokes the given callable object \texttt{f} with the content of the sequenced type \texttt{t} (tuples, pairs)

\textbf{Return} The result of the callable object when it’s called with the content of the given sequenced type.

\textbf{Note} This function is similar to \texttt{std::apply} (C++17)

\textbf{Parameters}

\begin{itemize}
\item \texttt{f}: Must be a callable object. If \texttt{f} is a member function pointer, the first argument in the sequenced type will be treated as the callee (this object).
\item \texttt{t}: A type whose contents are accessible through a call to \texttt{hpx::get}.
\end{itemize}

\textbf{Exceptions}

\begin{itemize}
\item \texttt{std::exception}: like objects thrown by call to object \texttt{f} with the arguments contained in the sequenceable type \texttt{t}.
\end{itemize}

```cpp
template<typename R, typename F, typename Tuple>
constexpr R invoke_fused_r (F &&f, Tuple &&t)
```

\textbf{Template Parameters}

\begin{itemize}
\item \texttt{R}: The result type of the function when it’s called with the content of the given sequenced type.
\end{itemize}

namespace hpx

namespace util
Functions

```cpp
template<typename M, typename C>
constexpr detail::mem_fn<M C::*> mem_fn(M C::* pm)
```

```cpp
template<typename R, typename C, typename ...Ps>
constexpr detail::mem_fn<R (C::*)(Ps...) const mem_fn R (C::*)(Ps...) const
```

```cpp
namespace hpx

namespace serialization

Functions

```cpp
template<typename Archive, typename F>
void serialize(Archive & ar, ::hpx::util::detail::one_shot_wrapper<F> & one_shot_wrapper, unsigned int const version = 0)
```

```cpp
namespace util

Functions

```cpp
template<typename F>
constexpr detail::one_shot_wrapper<typename std::decay<F>::type> one_shot(F &&f)
```

```cpp
namespace hpx

namespace util

Functions

```cpp
template<typename T>
std::enable_if<traits::is_bind_expression<typename std::decay<T>::type>::value, detail::protected_bind<typename std::decay<T>::type>>::type
```

```cpp
template<typename T>
std::enable_if<!traits::is_bind_expression<typename std::decay<T>::type>::value, T>::type protect(T &&v)
```
Defines

HPX_UTIL_REGISTER_UNIQUE_FUNCTION_DECLARATION (Sig, F, Name)
HPX_UTIL_REGISTER_UNIQUE_FUNCTION (Sig, F, Name)

namespace hpx

namespace util

Typedefs

template<typename Sig>
using unique_function_nonser = unique_function<Sig, false>

template<typename R, typename ...Ts, bool Serializable>
class unique_function<R (Ts...), Serializable> : public detail::basic_function<R
Ts..., false, Serializable>

Public Types

typedef R result_type

Public Functions

constexpr unique_function (std::nullptr_t = nullptr)
unique_function (unique_function&&)
unique_function &operator= (unique_function&&)

template<typename F, typename FD = typename std::decay<F>::type, typename Enable1 = typename std::enable_if<!
std::is_same<FD, unique_function>::value>::type, typename Enable2 = typename std::enable_if<
is_invocable_r_v<R, FD &, Ts...>>::type>
unique_function (F &&f)

Private Types

template<>
using base_type = detail::basic_function<R (Ts...) , false, Serializable>

template<typename R, typename Obj, typename ...Ts>
struct get_function_address<R (Obj::*) (Ts...) >
Public Static Functions

```cpp
static std::size_t call (R (Obj::* f)) Ts...
```

```cpp
template<typename R, typename Obj, typename ...Ts>
struct get_function_address<R (Obj::*) (Ts...) const>
```

Public Static Functions

```cpp
static constexpr std::size_t call (R (Obj::* f)) Ts...
    const
```

```cpp
namespace hpx
```

```cpp
namespace traits
```

```cpp
    template<typename F, typename Enable = void>
    struct get_function_address
```

Public Static Functions

```cpp
static constexpr std::size_t call (F const & f)
```

```cpp
    template<typename R, typename ...Ts>
    struct get_function_address<R (*)(Ts...) >
```

Public Static Functions

```cpp
static constexpr std::size_t call (R (*)(f)) Ts...
```

```cpp
    template<typename R, typename Obj, typename ...Ts>
    struct get_function_address<R (Obj::*) (Ts...) const>
```

Public Static Functions

```cpp
static std::size_t call (R (Obj::* f)) Ts...
    const
```

```cpp
    template<typename R, typename Obj, typename ...Ts>
    struct get_function_address<R (Obj::*) (Ts...) >
```
Public Static Functions

```cpp
static std::size_t call (R (Obj::* f)) Ts...
```

namespace hpx

namespace traits

```cpp
template<typename F, typename Enable = void>
struct get_function_annotation
```

Public Static Functions

```cpp
static constexpr char const *call (F const &)
```

namespace hpx

namespace traits

Variables

```cpp
template<typename T>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_action_v = is_action<T>::value
```

```cpp
template<typename T>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_bound_action_v = is_bound_action<T>::value
```

namespace hpx

namespace traits

Variables

```cpp
template<typename T>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_bind_expression_v = is_bind_expression<T>::value
```

```cpp
struct is_bind_expression : public std::is_bind_expression<T>
```

Subclassed by hpx::traits::is_bind_expression< T const >

namespace hpx

namespace traits

```cpp
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>.
futures

The contents of this module can be included with the header `hpx/modules/futures.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/futures.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

`HPX_MAKE_EXCEPTIONAL_FUTURE (T, errorcode, f, msg)`

namespace hpx

namespace lcos

Functions

template<
  typename R,
  typename U>
  hpx::lcos::future<R> make_future (hpx::lcos::future<U> &&f)

  template<
    typename R,
    typename U, 
    typename Conv>
  hpx::lcos::future<R> make_future (hpx::lcos::future<U> &&f, Conv &&conv)

  template<
    typename R, 
    typename U>
  hpx::lcos::future<R> make_future (hpx::lcos::shared_future<U> &f)

  template<
    typename R, 
    typename U, 
    typename Conv>
  hpx::lcos::future<R> make_future (hpx::lcos::shared_future<U> const &f, Conv &&conv)

  template<
    typename R>
  hpx::lcos::shared_future<R> make_shared_future (hpx::lcos::future<R> &&f)

  template<
    typename R>
  hpx::lcos::shared_future<R> &make_shared_future (hpx::lcos::shared_future<R> &f)

  template<
    typename R>
  hpx::lcos::shared_future<R> &&make_shared_future (hpx::lcos::shared_future<R> &&f)

  template<
    typename R>
  hpx::lcos::shared_future<R> const &make_shared_future (hpx::lcos::shared_future<R> const &f)

  template<
    typename T, 
    typename Allocator, 
    typename ...Ts>
  std::enable_if<
    std::is_constructible<T, Ts&&...>::value ||
    std::is_void<T>::value, future<T>>::type make_ready_future_alloc(
    Allocator const &a,
    Ts&&... ts)

  template<
    typename T, 
    typename ...Ts>
  std::enable_if<
    std::is_constructible<T, Ts&&...>::value ||
    std::is_void<T>::value, future<T>>::type make_ready_future(
    Ts&&... ts)

  template<
    int DeductionGuard = 0, 
    typename Allocator, 
    typename T>
future<typename hpx::util::decay_unwrap<T>::type> make_ready_future_alloc (Allocator const &a, T &&init)

template<int DeductionGuard = 0, typename T>
future<typename hpx::util::decay_unwrap<T>::type> make_ready_future (T &&init)

template<typename T>
future<T> make_exceptional_future (std::exception_ptr const &e)

template<typename T, typename E>
future<T> make_exceptional_future (E e)

template<int DeductionGuard = 0, typename T>
future<typename hpx::util::decay_unwrap<T>::type> make_ready_future_at (hpx::chrono::steady_time_point const &abs_time, T &&init)

template<int DeductionGuard = 0, typename T>
future<typename hpx::util::decay_unwrap<T>::type> make_ready_future_after (hpx::chrono::steady_duration const &rel_time, T &&init)

template<typename Allocator>
future<void> make_ready_future_alloc (Allocator const &a)

future<void> make_ready_future ()

future<void> make_ready_future_at (hpx::chrono::steady_time_point const &abs_time)

future<void> make_ready_future_after (hpx::chrono::steady_duration const &rel_time)

template<typename R>
class future : public hpx::lcos::detail::future_base<future<R>, R>

Public Types

template<>
using result_type = R

template<>
using shared_state_type = typename base_type::shared_state_type

Public Functions

future ()

future (future &&other)

future (future<future> &&other)

future (future<shared_future<R>> &&other)
template<typename T>
future(future<T> &&other, typename std::enable_if<std::is_void<R>::value && !traits::is_future<T>::value, T>::type* = nullptr)

future() 
future &operator=(future &&other)

shared_future<R> share() 

hpx::traits::future_traits<future>::result_type get()

hpx::traits::future_traits<future>::result_type get(error_code &ec)

template<typename F>
decltype(auto) then (F &&f, error_code &ec = throws)

template<typename T0, typename F>
decltype(auto) then (T0 &&t0, F &&f, error_code &ec = throws)

template<typename Allocator, typename F>
auto then_alloc (Allocator const &alloc, F &&f, error_code &ec = throws)

Private Types

template<>
using base_type = detail::future_base<future<R>, R>

Private Functions

future(hpx::intrusive_ptr<shared_state_type> const &state)
future(hpx::intrusive_ptr<shared_state_type> &&&state)

template<typename SharedState>
future(hpx::intrusive_ptr<SharedState> const &state)

Friends

friend hpx::lcos::hpx::traits::future_access

template<typename Receiver>
detail::operation_state<Receiver, future> tag_invoke (hpx::execution::experimental::connect_t, future &&f, Receiver &&receiver)

struct invalidate
Public Functions

```cpp
template<>
invalidate (future &f)

template<>
~invalidate ()
```

Public Members

```cpp
template<>
future &f_
```

```cpp
class shared_future : public hpx::lcos::detail::future_base<shared_future<R>, R>
```

Public Types

```cpp
template<>
using result_type = R
```

```cpp
template<>
using shared_state_type = typename base_type::shared_state_type
```

Public Functions

```cpp
shared_future ()
shared_future (shared_future const &other)
shared_future (shared_future &&other)
shared_future (future<R> &&other)
shared_future (future<shared_future> &&other)
```

```cpp
template<typename T>
shared_future (shared_future<T> const &other, typename std::enable_if<
  std::is_void<R>::value && !traits::is_future<T>::value, T>::type* = nullptr)
```

```cpp
~shared_future ()
```

```cpp
shared_future &operator= (shared_future const &other)
shared_future &operator= (shared_future &&other)
```

```cpp
hpx::traits::future_traits<shared_future>::result_type get () const
```

```cpp
hpx::traits::future_traits<shared_future>::result_type get (error_code &ec) const
```

```cpp
template<type R>
dectype(auto) then (F &&f, error_code &ec = throws) const
```

```cpp
template<type R, type T0, typename F>
dectype(auto) then (T0 &&t0, F &&f, error_code &ec = throws) const
```
template<typename Allocator, typename F>
auto then_alloc (Allocator const &alloc, F &&f, error_code &ec = throws)

Private Types

typedef detail::future_base<shared_future<R>, R> base_type

Private Functions

shared_future (hpx::intrusive_ptr<shared_state_type> const &state)
shared_future (hpx::intrusive_ptr<shared_state_type> &state)
template<typename SharedState>
shared_future (hpx::intrusive_ptr<SharedState> const &state)

Friends

template<typename Receiver>
detail::operation_state<Receiver, shared_future> tag_invoke (hpx::execution::experimental::connect_t, shared_future &&f, Receiver &&&receiver)

template<typename Receiver>
detail::operation_state<Receiver, shared_future> tag_invoke (hpx::execution::experimental::connect_t, shared_future &f, Receiver &&&receiver)

namespace serialization

Functions

template<typename Archive, typename T>
void serialize (Archive &ar, ::hpx::lcos::future<T> &f, unsigned version)

namespace hpx

namespace lcos

namespace local

template<typename Result, bool Cancelable>
class futures_factory<Result, Cancelable>
Public Functions

futures_factory()

template<typename Executor, typename F>
futures_factory (Executor & exec, F && f)

template<typename Executor>
futures_factory (Executor & exec, Result (* f))

template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, futures_factory>::value>::type>
futures_factory (F && f)

futures_factory (Result (* f))

~futures_factory()

futures_factory (futures_factory const & rhs)

futures_factory & operator= (futures_factory const & rhs)

futures_factory (futures_factory && rhs)

futures_factory & operator= (futures_factory && rhs)

void operator() () const

threads::thread_id_ref_type apply (const char * annotation = "futures_factory::apply",
launch policy = launch::async, error_code & ec = throws) const

threads::thread_id_ref_type apply (threads::thread_pool_base * pool, const char * annotation = "futures_factory::apply", launch policy = launch::async, error_code & ec = throws) const

lcos::future<Result> get_future (error_code & ec = throws)

bool valid() const

void set_exception (std::exception_ptr const & e)

Protected Types

typedef lcos::detail::task_base<Result> task_impl_type

Protected Attributes

hpx::intrusive_ptr<task_impl_type> task_

bool future_obtained_

namespace hpx

namespace lcos
namespace local

template<typename R, typename ...Ts>
class packaged_task<R(Ts...)>

Public Functions

packaged_task()

template<typename F, typename FD = std::decay_t<F>, typename Enable = std::enable_if_t<!std::is_same_v<FD, packaged_task>>, 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>> &a, F &f)

packaged_task (packaged_task const &rhs)

packaged_task (packaged_task &&rhs)

packaged_task &operator= (packaged_task const &rhs)

packaged_task &operator= (packaged_task &&rhs)

void swap (packaged_task &rhs)

void operator() (Ts... ts)

lcos::future<R> get_future (error_code &ec = throws)

bool valid() const

void reset (error_code &ec = throws)

void set_exception (std::exception_ptr const &e)

Private Types

template<>
using function_type = util::unique_function_nonser<R(Ts...)>

Private Members

function_type function_

local::promise<R> promise_

namespace std
Functions

template<typename Sig>
void swap (hpx::lcos::local::packaged_task<Sig> &lhs, hpx::lcos::local::packaged_task<Sig> &rhs)

namespace hpx

namespace lcos

namespace local

Functions

template<typename R>
void swap (promise<R> &x, promise<R> &y)

template<typename R>
class promise : public hpx::lcos::local::detail::promise_base<R>

Public Functions

promise ()
template<typename Allocator>
promise (std::allocator_arg_t, Allocator const &a)
promise (promise &&other)
~promise ()
promise &operator= (promise &&other)
void swap (promise &other)
void setValue (R const &r)
void setValue (R &&r)
template<typename ... Ts>
void setValue (Ts &... ts)

Private Types

template<>
using base_type = detail::promise_base<R>

template<typename R>
class promise<R&> : public hpx::lcos::local::detail::promise_base<R&>
Public Functions

promise()

template<
typename Allocator>
promise (std::allocator_arg_t, Allocator const &a)

promise (promise &&other)

~promise()

promise &operator= (promise &&other)

void swap (promise &other)

void set_value (R &r)

Private Types

template<>
using base_type = detail::promise_base<R&>

template<>
class promise<void>: public hpx::lcos::local::detail::promise_base<void>

Public Functions

promise()

template<
typename Allocator>
promise (std::allocator_arg_t, Allocator const &a)

promise (promise &&other)

~promise()

promise &operator= (promise &&other)

void swap (promise &other)

void set_value ()

Private Types

template<>
using base_type = detail::promise_base<void>

namespace hpx

namespace traits

struct acquire_future_disp
Public Functions

template<typename T>
acquire_future<T>::type operator() (T &&t) const

namespace hpx

namespace traits

struct acquire_shared_state_disp

Public Functions

template<typename T>
acquire_shared_state<T>::type operator() (T &&t) const

template<typename R>
struct future_access<lcos::future<R> >

Public Static Functions

template<typename SharedState>
static lcos::future<R> create (hpx::intrusive_ptr<SharedState> const &shared_state)

template<typename T = void>
static lcos::future<R> create (detail::shared_state_ptr_for_t<lcos::future<lcos::future<R>>>> const &shared_state)

template<typename SharedState>
static lcos::future<R> create (hpx::intrusive_ptr<SharedState> &&&shared_state)

template<typename T = void>
static lcos::future<R> create (detail::shared_state_ptr_for_t<lcos::future<lcos::future<R>>>> &&&shared_state)

template<typename SharedState>
static lcos::future<R> create (SharedState *shared_state, bool addref = true)

static traits::detail::shared_state_ptr_t<R> const &get_shared_state (lcos::future<R> const &f)

static traits::detail::shared_state_ptr_t<R>::element_type *detach_shared_state (lcos::future<R> &&&f)

template<typename Destination>
static void transfer_result (lcos::future<R> &&&src, Destination &dest)
Private Static Functions

template<typename Destination>
static void transfer_result_impl (lcos::future<R> &&src, Destination &dest, std::false_type)
Public Static Functions

```cpp
template<typename SharedState>
static lcos::future<R> create (hpx::intrusive_ptr<SharedState> const &shared_state)

template<typename T = void>
static lcos::future<R> create (detail::shared_state_ptr_for_t<lcos::future<lcos::future<R>>> const &shared_state)

template<typename SharedState>
static lcos::future<R> create (hpx::intrusive_ptr<SharedState> &&shared_state)

template<typename T = void>
static lcos::future<R> create (detail::shared_state_ptr_for_t<lcos::future<lcos::future<R>>> &&shared_state)

template<typename SharedState>
static lcos::future<R> create (SharedState *shared_state, bool addref = true)

static traits::detail::shared_state_ptr_t<R> const &get_shared_state (lcos::future<R> const &f)

static traits::detail::shared_state_ptr_t<R>::element_type *detach_shared_state (lcos::future<R> &&f)

template<typename Destination>
static void transfer_result (lcos::future<R> &&&src, Destination &dest)
```

Private Static Functions

```cpp
template<typename Destination>
static void transfer_result_impl (lcos::future<R> &&&src, Destination &dest, std::false_type)

template<typename Destination>
static void transfer_result_impl (lcos::future<R> &&&src, Destination &dest, std::true_type)
```

```cpp
template<typename R>
struct future_access<lcos::shared_future<R>>
```

Public Static Functions

```cpp
template<typename SharedState>
static lcos::shared_future<R> create (hpx::intrusive_ptr<SharedState> const &shared_state)

template<typename T = void>
static lcos::shared_future<R> create (detail::shared_state_ptr_for_t<lcos::shared_future<lcos::future<R>>> const &shared_state)

template<typename SharedState>
static lcos::shared_future<R> create (hpx::intrusive_ptr<SharedState> &&shared_state)

template<typename T = void>
static lcos::shared_future<R> create (detail::shared_state_ptr_for_t<lcos::shared_future<lcos::future<R>>> &&shared_state)
```
template<typename SharedState>
static lcos::shared_future<R> create (SharedState *shared_state, bool addref = true)

static traits::detail::shared_state_ptr_t<R> const &get_shared_state (lcos::shared_future<R> const &f)

static traits::detail::shared_state_ptr_t<R>::element_type *detach_shared_state (lcos::shared_future<R> const &f)

template<typename Destination>
static void transfer_result (lcos::shared_future<R> &&src, Destination &dest)

Private Static Functions

template<typename Destination>
static void transfer_result_impl (lcos::shared_future<R> &&src, Destination &dest, std::false_type)

template<typename Destination>
static void transfer_result_impl (lcos::shared_future<R> &&src, Destination &dest, std::true_type)

namespace hpx

namespace traits

Typedefs

template<typename Future, typename F>
using future_then_result_t = typename future_then_result<Future, F>::type

template<typename R>
struct future_traits<lcos::future<R>>

Public Types

template<>
using type = R

template<>
using result_type = R

template<typename R>
struct future_traits<lcos::shared_future<R>>
Public Types

template<>
using type = R

template<>
using result_type = R const&

template<>
struct future_traits<lcos::shared_future<void>>

Public Types

template<>
using type = void

template<>
using result_type = void

namespace hpx

namespace traits

Typedefs

template<typename Future>
using future_traits_t = typename future_traits<Future>::type

template<typename R>
struct future_traits<lcos::future<R>>

Public Types

template<>
using type = R

template<>
using result_type = R

template<typename R>
struct future_traits<lcos::shared_future<R>>

Public Types

template<>
using type = R

template<>
using result_type = R const&

template<>
struct future_traits<lcos::shared_future<void>>
Public Types

template<>
using type = void
template<>
using result_type = void

namespace hpx

namespace traits

template<typename Result, typename RemoteResult, typename Enable = void>
struct get_remote_result

Public Static Functions

static Result call (RemoteResult const &rhs)
static Result call (RemoteResult &&rhs)
template<typename Result>
struct get_remote_result<Result, Result>

Public Static Functions

static Result const &call (Result const &rhs)
static Result &&call (Result &&rhs)

namespace hpx

namespace traits

Variables

template<typename R>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_future_v = is_future<R>::value

template<typename R>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_ref_wrapped_future_v=is_ref_wrapped_future<R>::value

template<typename Future>
struct is_future : public hpx::traits::detail::is_future_customization_point<Future>
    Subclassed by hpx::traits::is_ref_wrapped_future< std::reference_wrapper< Future > >

namespace hpx

namespace traits

template<typename R>
struct future_range_traits<R, true>
Public Types

typedef range_traits<R>::value_type future_type

template<>
struct promise_local_result<util::unused_type>

namespace hpx

namespace traits

Typedefs

template<typename Result>
using promise_local_result_t = typename promise_local_result<Result>::type

template<typename Result, typename Enable = void>
struct promise_local_result

namespace hpx

namespace traits

Typedefs

template<typename Result>
using promise_remote_result_t = typename promise_remote_result<Result>::type

template<typename Result, typename Enable = void>
struct promise_remote_result
Public Types

typedef Result type

template<>
struct promise_remote_result<void>

Public Types

typedef hpx::util::unused_type type

hardware

The contents of this module can be included with the header hpx/modules/hardware.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/hardware.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace util

namespace hardware

Functions

template<typename T, typename U>
bool has_bit_set(T value, U bit)

template<std::size_t N, typename T>
T unbounded_shl(T x)

template<std::size_t N, typename T>
T unbounded_shr(T x)

template<std::size_t Low, std::size_t High, typename Result, typename T>
Result get_bit_range(T x)

template<std::size_t Low, typename Result, typename T>
Result pack_bits(T x)

template<std::size_t N, typename T>
struct unbounded_shifter
Public Static Functions

```cpp
static T shl (T x)
static T shr (T x)
```

```
namespace hpx

namespace util

namespace hardware

Functions

void cpuid (std::uint32_t (&cpuinfo)[4], std::uint32_t eax)
void cpuidex (std::uint32_t (&cpuinfo)[4], std::uint32_t eax, std::uint32_t ecx)
```

Public Types

```
enum info
Values:
    eax = 0
    ebx = 1
    ecx = 2
    edx = 3
```

```
namespace hpx

namespace util

namespace hardware
```
namespace hpx

namespace util

namespace hardware

Functions

std::uint64_t hpx::util::hardware::timestamp_cuda()

namespace hpx

namespace util

namespace hardware

Functions

std::uint64_t timestamp()

hashing

The contents of this module can be included with the header hpx/modules/hashing.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/hashing.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace util

Functions

template<
std::uint64_t N>
constexpr std::uint64_t fibhash(std::uint64_t i)

namespace hpx

namespace util

class jenkins_hash

#include <jenkins_hash.hpp> The jenkins_hash class encapsulates a hash calculation function published by Bob Jenkins here: http://burtleburtle.net/bob/hash

Public Types

enum seedenum

The seedenum is used as a dummy parameter to distinguish the different constructors

Values:

seed = 1

typedef std::uint32_t size_type
	his is the type representing the result of this hash
Public Functions

`jenkins_hash()`
constructors and destructor

`jenkins_hash(size_type size)`

`jenkins_hash(size_type seedval, seedenum)`

`~jenkins_hash()`

`size_type operator()(std::string const &key) const`
calculate the hash value for the given key

`size_type operator()(char const *key) const`

`bool reset(size_type size)`
re-seed the hash generator

`void set_seed(size_type seedval)`
initialize the hash generator to a specific seed

`void swap(jenkins_hash &rhs)`
support for std::swap

Protected Functions

`size_type hash(const char *, std::size_t length) const`

Private Members

`size_type seed_`

include_local

The contents of this module can be included with the header `hpx/modules/include_local.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/include_local.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

Typedefs

```cpp
template<typename OnCompletion = lcos::local::detail::empty_oncompletion>
using barrier = lcos::local::cpp20_barrier<OnCompletion>
```
Typedefs

using latch = hpx::lcos::local::cpp20_latch

namespace hpx

Typedefs

template<std::ptrdiff_t LeastMaxValue = std::PTRDIFF_MAX>
using counting_semaphore = hpx::lcos::local::cpp20_counting_semaphore<LeastMaxValue>

using binary_semaphore = hpx::lcos::local::cpp20_binary_semaphore<>

namespace hpx

Typedefs

using task_cancelled_exception = hpx::parallel::task_canceled_exception

ini

The contents of this module can be included with the header hpx/modules/ini.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/ini.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

HPX_SECTION_VERSION

namespace hpx

namespace util

class section

Public Types

typedef util::function_nonser<void (std::string const&, std::string const&)> entry_changed_func
typedef std::pair<std::string, entry_changed_func> entry_type
typedef std::map<std::string, entry_type> entry_map
typedef std::map<std::string, section> section_map
Public Functions

section()
section (std::string const &filename, section *root = nullptr)
section (section const &in)
~section()
section &operator=(section const &rhs)
void parse (std::string &sourcename, std::vector<std::string> const &lines, bool verify_existing = true, bool weed_out_comments = true, bool replace_existing = true)
void parse (std::string const &sourcename, std::string const &line, bool verify_existing = true, bool weed_out_comments = true, bool replace_existing = true)
void read (std::string const &filename)
void merge (std::string const &second)
void merge (section const &second)
void dump (int ind = 0) const
void dump (int ind, std::ostream &strm) const
void add_section (std::string const &sec_name, section &sec, section *root = nullptr)
section *add_section_if_new (std::string const &sec_name)
bool has_section (std::string const &sec_name) const
section *get_section (std::string const &sec_name)
section const *get_section (std::string const &sec_name) const
section_map &get_sections ()
section_map const &get_sections () const
void add_entry (std::string const &key, entry_type const &val)
void add_entry (std::string const &key, std::string const &val)
bool has_entry (std::string const &key) const
std::string get_entry (std::string const &key) const
std::string get_entry (std::string const &key, std::string const &dfltr) const
template<typename T>
std::string get_entry (std::string const &key, T dfltr) const
void add_notification_callback (std::string const &key, entry_changed_func const &callback)
entry_map const &get_entries () const
std::string expand (std::string const &str) const
void `expand` (std::string &str, std::string::size_type len) const
void `set_root` (section *, bool recursive = false)

section *`get_root` () const
std::string `get_name` () const
std::string `get_parent_name` () const
std::string `get_full_name` () const
void `set_name` (std::string const &name)

Protected Functions

void `line_msg` (std::string msg, std::string const &file, int lnum = 0, std::string const &line = "")

section &`clone_from` (section const &rhs, section *root = nullptr)

Private Types

using mutex_type = util::spinlock

Private Functions

section *`this_` ()

template<typename Archive>
void `save` (Archive &ar, const unsigned int version) const

template<typename Archive>
void `load` (Archive &ar, const unsigned int version)

void `add_section` (std::unique_lock<mutex_type> &l, std::string const &sec_name, section &sec, section *root = nullptr)
bool `has_section` (std::unique_lock<mutex_type> &l, std::string const &sec_name)

section *`get_section` (std::unique_lock<mutex_type> &l, std::string const &sec_name)

section const *`get_section` (std::unique_lock<mutex_type> &l, std::string const &sec_name) const

section *`add_section_if_new` (std::unique_lock<mutex_type> &l, std::string const &sec_name)

void `add_entry` (std::unique_lock<mutex_type> &l, std::string const &fullkey, std::string const &key, std::string val)
void `add_entry` (std::unique_lock<mutex_type> &l, std::string const &fullkey, std::string const &key, entry_type const &val)
bool `has_entry` (std::unique_lock<mutex_type> &l, std::string const &key) const

std::string `get_entry` (std::unique_lock<mutex_type> &l, std::string const &key) const
std::string get_entry(std::unique_lock<mutex_type> &l, std::string const &key, std::string const &dflt) const

void add_notification_callback(std::unique_lock<mutex_type> &l, std::string const &key, entry_changed_func const &callback)

std::string expand(std::unique_lock<mutex_type> &l, std::string const &in) const

void expand(std::unique_lock<mutex_type> &l, std::string &out, std::string::size_type &length) const

void expand_bracket(std::unique_lock<mutex_type> &l, std::string &out, std::string::size_type &length) const

void expand_brace(std::unique_lock<mutex_type> &l, std::string &out, std::string::size_type &length) const

std::string expand_only(std::unique_lock<mutex_type> &l, std::string const &in, std::string const &expand_this) const

void expand_only(std::unique_lock<mutex_type> &l, std::string &out, std::string::size_type &length, std::string const &expand_this) const

void expand_bracket_only(std::unique_lock<mutex_type> &l, std::string::size_type &length, std::string const &expand_this) const

void expand_brace_only(std::unique_lock<mutex_type> &l, std::string::size_type &length, std::string const &expand_this) const

**Private Members**

section *root_

entry_map entries_

section_map sections_

std::string name_

std::string parent_name_

mutex_type mtx_

**Friends**

friend hpx::util::hpx::serialization::access

**init_runtime_local**

The contents of this module can be included with the header hpx/modules/init_runtime_local.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/init_runtime_local.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.
Defines

HPX_PREFIX
HPX_APPLICATION_STRING

namespace hpx

namespace local

Functions

int init (std::function<int> hpx::program_options::variables_map&
    > f, int argc, char **argv, init_params const &params = init_params()

int init (std::function<int> int, char**
    > f, int argc, char **argv, init_params const &params = init_params()

int init (std::function<int>
    > f, int argc, char **argv, init_params const &params = init_params()

int init (std::nullptr_t, int argc, char **argv, init_params const &params = init_params())

bool start (std::function<int> hpx::program_options::variables_map&
    > f, int argc, char **argv, init_params const &params = init_params()

bool start (std::function<int> int, char**
    > f, int argc, char **argv, init_params const &params = init_params()

bool start (std::function<int>
    > f, int argc, char **argv, init_params const &params = init_params()

bool start (std::nullptr_t, int argc, char **argv, init_params const &params = init_params())

int finalize (error_code &ec = throws)

int stop (error_code &ec = throws)

int suspend (error_code &ec = throws)

int resume (error_code &ec = throws)

struct init_params

Public Members

std::reference_wrapper<hpx::program_options::options_description const> desc_cmdline = detail::default_desc
std::vector<std::string> cfg
startup_function_type startup
shutdown_function_type shutdown
hpx::resource::partitioner_mode rp_mode = hpx::resource::mode_default
hpx::local::detail::rp_callback_type rp_callback
io_service

The contents of this module can be included with the header \texttt{hpx/modules/io-service.hpp}. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header \texttt{hpx/modules/io-service.hpp}, not the particular header in which the functionality you would like to use is defined. See \textit{Public API} for a list of names that are part of the public HPX API.

\begin{verbatim}
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)

        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
        • start_thread: [in]

        io_service_pool (threads::policies::callback_notifier const &notifier, char const *pool_name = "", char const *name_postfix = ")
        Construct the io_service pool.

        Parameters
        • start_thread: [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.
\end{verbatim}
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

std::size_t size() const
    Get number of threads associated with this I/O service.

void thread_run(std::size_t index, barrier *startup = nullptr)
    Activate the thread index for this thread pool.

char const *get_name() const
    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 = asio::io_context::work

Private Functions

work_type initialize_work(asio::io_context &io_service)

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.
HPX Documentation, master

`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_`

**iterator_support**

The contents of this module can be included with the header `hpx/modules/iterator_support.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/iterator_support.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

**Defines**

`HPX_ITERATOR_TRAVERSAL_TAG_NS`

`namespace hpx`

```cpp
namespace iterators {

struct bidirectional_traversal_tag : public hpx::iterators::forward_traversal_tag {
    // Subclassed by hpx::iterators::random_access_traversal_tag
}

struct forward_traversal_tag : public hpx::iterators::single_pass_traversal_tag {
    // Subclassed by hpx::iterators::bidirectional_traversal_tag
}

struct incrementable_traversal_tag : public hpx::iterators::no_traversal_tag {
    // Subclassed by hpx::iterators::single_pass_traversal_tag
}

struct no_traversal_tag {
    // Subclassed by hpx::iterators::incrementable_traversal_tag
}

struct single_pass_traversal_tag : public hpx::iterators::incrementable_traversal_tag {
    // Subclassed by hpx::iterators::forward_traversal_tag
}

namespace traits {
```

2.8. API reference 961
**Typedefs**

```cpp
template<typename Traversal>
using pure_traversal_tag = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::pure_traversal_tag<Traversal>::type

template<typename Traversal>
using pure_traversal_tag_t = typename pure_traversal_tag<Traversal>::type

template<typename Iterator>
using pure_iterator_traversal = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::pure_iterator_traversal<Iterator>::type

template<typename Iterator>
using pure_iterator_traversal_t = typename pure_iterator_traversal<Iterator>::type

template<typename Cat>
using iterator_category_to_traversal = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::iterator_category_to_traversal<Cat>::type

template<typename Cat>
using iterator_category_to_traversal_t = typename iterator_category_to_traversal<Cat>::type

template<typename Iterator>
using iterator_traversal = HPX_ITERATOR_TRAVERSAL_TAG_NS::iterators::iterator_traversal<Iterator>::type

template<typename Iterator>
using iterator_traversal_t = typename iterator_traversal<Iterator>::type
```

**Variables**

```cpp
template<typename Traversal>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::pure_traversal_tag_v = pure_traversal_tag<Traversal>::value

template<typename Iterator>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::pure_iterator_traversal_v = pure_iterator_traversal<Iterator>::value

template<typename Cat>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::iterator_category_to_traversal_v = iterator_category_to_traversal<Cat>::value

template<typename Iterator>
HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::iterator_traversal_v = iterator_traversal<Iterator>::value
```

**Public Functions**

```cpp
counting_iterator()
counting_iterator(counting_iterator &&rhs)
counting_iterator & operator=(counting_iterator &&rhs)
counting_iterator(const counting_iterator &rhs)
counting_iterator & operator=(const counting_iterator &rhs)
counting_iterator(Incrementable x)
```
Private Types

template<>
using base_type = typename detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>::type

Private Functions

template<typename Iterator>
bool equal (Iterator const &rhs) const

void increment ()

void decrement ()

template<typename Distance>
void advance (Distance n)

base_type::reference dereference () const

template<typename OtherIncrementable>
base_type::difference_type distance_to (counting_iterator<OtherIncrementable, CategoryOrTraversal, Difference> const &y) const

Friends

friend iterator_core_access

namespace hpx

namespace util

Functions

template<typename Incrementable>
counting_iterator<Incrementable> make_counting_iterator (Incrementable x)

template<typename Incrementable, typename CategoryOrTraversal, typename Difference, typename Enable>
class counting_iterator

Public Functions

counting_iterator ()

counting_iterator (counting_iterator &&rhs)

counting_iterator &operator= (counting_iterator &&rhs)

counting_iterator (counting_iterator const &rhs)

counting_iterator &operator= (counting_iterator const &rhs)

counting_iterator (Incrementable x)
Private Types

```cpp
template<>
using base_type = typename detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>::type
```

Private Functions

```cpp
base_type::reference dereference() const
```

Friends

```cpp
friend hpx::util::iterator_core_access
template<typename Incrementable, typename CategoryOrTraversal, typename Difference>
class counting_iterator<Incrementable, CategoryOrTraversal, Difference, typename std::enable_if<std::is_integral<Incrementable>::value>::type> :
public hpx::util::iterator_adaptor<counting_iterator<Incrementable, CategoryOrTraversal, Difference>, Incrementable, Incrementable, traversal, Incrementable const &, difference>
```

Public Functions

```cpp
counting_iterator()
counting_iterator(counting_iterator &&rhs)
counting_iterator &operator=(counting_iterator &&rhs)
counting_iterator(counting_iterator const &rhs)
counting_iterator &operator=(counting_iterator const &rhs)
counting_iterator(Incrementable x)
```

Private Types

```cpp
template<>
using base_type = typename detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>::type
```

Private Functions

```cpp
template<typename Iterator>
bool equal(Iterator const &rhs) const
void increment()
void decrement()
template<typename Distance>
void advance(Distance n)
base_type::reference dereference() const
template<typename OtherIncrementable>
base_type::difference_type distance_to(counting_iterator<OtherIncrementable, CategoryOrTraversal, Difference> const &y) const
```
Friends

friend hpx::util::iterator_core_access

namespace hpx

namespace util

Functions

template<typename Generator>
generator_iterator<Generator> make_generator_iterator (Generator &gen)

template<typename Generator>
generator_iterator (Generator *)

template<typename Generator>
class generator_iterator : public hpx::util::iterator_facade<generator_iterator<Generator>, Generator::result_type, std::forward_iterator_tag, Generator::result_type const

Public Functions

generator_iterator ()

generator_iterator (Generator *g)

void increment ()

Generator::result_type const &dereference () const

bool equal (generator_iterator const &y) const

Private Types

template<>
using base_type = iterator_facade<generator_iterator<Generator>, typename Generator::result_type, std::forward_iterator_tag, typename Generator::result_type const

Private Members

Generator *m_g

Generator::result_type m_value

namespace hpx

namespace util

template<typename Derived, typename Base, typename Value = void, typename Category = void, typename Reference = void, typename Difference = void, typename Pointer = void>
class iterator_adaptor : public hpx::util::iterator_facade<Derived, value_type, iterator_category, reference_type, difference_type, void>

Subclassed by hpx::util::counting_iterator< Incrementable, CategoryOrTraversal, Difference, typename std::enable_if< std::is_integral< Incrementable >::value >::type >
Public Types

template<>
using base_type = Base

Public Functions

iterator_adaptor()
iterator_adaptor(Base const &iter)
Base const &base() const

Protected Types

template<>
using base_adaptor_type = typename hpx::util::detail::iterator_adaptor_base<Derived, Base, Value, Category, Reference, Difference, Pointer>::type
typedef iterator_adaptor<Derived, Base, Value, Category, Reference, Difference, Pointer> iterator_adaptor_

Protected Functions

Base const &base_reference() const
Base &base_reference()

Private Functions

base_adaptor_type::reference dereference() const
template<typename OtherDerived, typename OtherIterator, typename V, typename C, typename R, typename D, typename P>
bool equal (iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D, P> const &x) const
template<typename DifferenceType>
void advance (DifferenceType n)
void increment()

template<typename Iterator = Base, typename Enable = typename std::enable_if<std::is_bidirectional_iterator<Iterator>::value>::type>
void decrement()

template<typename OtherDerived, typename OtherIterator, typename V, typename C, typename R, typename P> base_adaptor_type::difference_type distance_to (iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D, P> const &y)
const

966 Chapter 2. What’s so special about HPX?
Private Members

Base `iterator_`

Friends

friend hpx::util::hpx::util::iterator_core_access

Defines

`HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD` (prefix, op, result_type)

`HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD_EX` (prefix, op, result_type, cond)

namespace hpx

namespace util

Functions

```cpp
template<typename Derived, typename T, typename Category, typename Reference, typename Distance, typename Pointer>
util::detail::postfix_increment_result_t<Derived, typename Derived::value_type, typename Derived::reference> operator++
```
```cpp
std::enable_if_t<std::is_same_v<typename Derived::iterator_category, std::random_access_iterator_tag>, Derived> operator+(iterator_facade<Derived, T, Category, Reference, Distance, Pointer> const &it, Derived::difference_type n)

class iterator_core_access

Public Static Functions

template<typename Iterator1, typename Iterator2>
static bool equal(Iterator1 const &lhs, Iterator2 const &rhs)

template<typename Iterator>
static void increment(Iterator &it)

template<typename Iterator>
static void decrement(Iterator &it)

template<typename Reference, typename Iterator>
static Reference dereference(Iterator const &it)
```
template<typename Iterator, typename Distance>
static void advance (Iterator &it, Distance n)

template<typename Iterator1, typename Iterator2>
static std::iterator_traits<Iterator1>::difference_type distance_to (Iterator1 const &lhs, Iterator2 const &rhs)

template<typename Derived, typename T, typename Category, typename Reference = T&, typename Distance = std::ptrdiff_t, typename Pointer = void>
struct iterator_facade : public hpx::util::detail::iterator_facade_base<Derived, T, Category, T&, std::ptrdiff_t, void>
Subclassed by hpx::util::iterator_adaptor< Derived, Base, Value, Category, Reference, Difference, Pointer >

Public Types

template<>
using iterator_category = typename base_type::iterator_category

template<>
using value_type = typename base_type::value_type

template<>
using difference_type = typename base_type::difference_type

template<>
using pointer = typename base_type::pointer

template<>
using reference = typename base_type::reference

Public Functions

iterator_facade()

Protected Types

template<>
using iterator_adaptor_ = iterator_facade<Derived, T, Category, Reference, Distance, Pointer>

Private Types

template<>
using base_type = detail::iterator_facade_base<Derived, T, Category, Reference, Distance, Pointer>

namespace hpx

namespace ranges

2.8. API reference
Typedefs

template<typename I, typename S = I>
using subrange_t = hpx::util::iterator_range<I, S>

namespace util

Functions

template<typename Range, typename Iterator = typename traits::range_iterator<Range>::type, typename Sentinel = typename traits::range_iterator<Range>::type>
std::enable_if<traits::is_range<Range>::value, iterator_range<Iterator, Sentinel>>::type make_iterator_range(Range &r)

template<typename Range, typename Iterator = typename traits::range_iterator<Range const>::type, typename Sentinel = typename traits::range_iterator<Range const>::type>
std::enable_if<traits::is_range<Range>::value, iterator_range<Iterator, Sentinel>>::type make_iterator_range(Range const &r)

template<typename Iterator, typename Sentinel = Iterator>
std::enable_if<traits::is_iterator<Iterator>::value, iterator_range<Iterator, Sentinel>>::type make_iterator_range(Iterator iterator, Sentinel sentinel)

class iterator_range

Public Functions

iterator_range()

iterator_range(Iterator iterator, Sentinel sentinel)

Iterator begin() const

Iterator end() const

std::ptrdiff_t size() const

bool empty() const
**Private Members**

Iterator _iterator

Sentinel _sentinel

namespace hpx

namespace util

namespace range_adl

**Functions**

template<typename C, typename Iterator = typename detail::iterator<C>::type>
constexpr Iterator begin (C &c)

template<typename C, typename Iterator = typename detail::iterator<C const>::type>
constexpr Iterator begin (C const &c)

template<typename C, typename Sentinel = typename detail::sentinel<C>::type>
constexpr Sentinel end (C &c)

template<typename C, typename Sentinel = typename detail::sentinel<C const>::type>
constexpr Sentinel end (C const &c)

template<typename C, typename Iterator = typename detail::iterator<C const>::type, typename Sentinel = typename detail::sentinel<C const>::type>
constexpr std::size_t size (C const &c)

template<typename C, typename Iterator = typename detail::iterator<C const>::type, typename Sentinel = typename detail::sentinel<C const>::type>
constexpr bool empty (C const &c)

namespace hpx

namespace util

**Functions**

template<typename Transformer, typename Iterator>
transform_iterator<Iterator, Transformer> make_transform_iterator (Iterator const &it, Transformer const &f)

template<typename Transformer, typename Iterator>
transform_iterator<Iterator, Transformer> make_transform_iterator (Iterator const &it)

template<typename Iterator, typename Transformer, typename Reference, typename Value, typename Category>
class transform_iterator
Public Functions

transform_iterator()
transform_iterator (Iterator const &it)
transform_iterator (Iterator const &it, Transformer const &f)

template<
typename OtherIterator, typename OtherTransformer, typename OtherReference, typename OtherValue, typename OtherCategory, typename OtherDifference>
    transform_iterator(transform_iterator<OtherIterator, OtherTransformer, OtherReference, OtherValue, OtherCategory, OtherDifference> const &t, transformer const &transformer () const

Private Types

typedef detail::transform_iterator_base<Iterator, Transformer, Reference, Value, Category, Difference>::type base_type

Private Functions

base_type::reference dereference () const

Private Members

Transformer transformer_

Friends

friend hpx::util::hpx::util::iterator_core_access

class zip_iterator<hpx::tuple<Ts...>> : public hpx::util::detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<hpx::tuple<Ts...>>>

Public Functions

zip_iterator()
zip_iterator (Ts const &... vs)
zip_iterator (hpx::tuple<Ts...> &&t)
zip_iterator (zip_iterator const &other)
zip_iterator (zip_iterator &&other)
zip_iterator & operator= (zip_iterator const &other)
zip_iterator & operator= (zip_iterator &&other)
template<typename ...Ts_>
std::enable_if<std::is_assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iterator_tuple_type>::value, zip_iterator&>::type
operator=(zip_iterator<Ts_...> const &other)

Private Types

template<>
using base_type = detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<hpx::tuple<Ts...>>>

template<typename F, typename ...Ts>
struct lift_zipped_iterators<F, util::zip_iterator<Ts...>>

Public Types

typedef util::zip_iterator<Ts...>::iterator_tuple_type tuple_type

typedef hpx::tuple<typename element_result_of<typename F::template apply<Ts>, Ts>::type...> result_type

Public Static Functions

template<std::size_t ...Is, typename ...Ts_>
static result_type call (util::index_pack<Is...>, hpx::tuple<Ts_...> const &t)

template<typename ...Ts_>
static result_type call (util::zip_iterator<Ts_...> const &iter)

namespace hpx

namespace traits

namespace functional

template<typename F, typename ...Ts>
struct lift_zipped_iterators<F, util::zip_iterator<Ts...>>

Public Types

typedef util::zip_iterator<Ts...>::iterator_tuple_type tuple_type

typedef hpx::tuple<typename element_result_of<typename F::template apply<Ts>, Ts>::type...> result_type
Public Static Functions

template<\typename Is, \ldots, Ts_>
static result_type call (util::index_pack<Is_\ldots>, hpx::tuple<Ts_\ldots> const \&t)

template<\typename Ts_>
static result_type call (util::zip_iterator<Ts_\ldots> const \&iter)

namespace util

Functions

template<\typename Ts_>
zip_iterator<typename std::decay<Ts_>::type_\ldots> make_zip_iterator (Ts_\&\ldots vs)

template<\typename Ts_>
class zip_iterator : public hpx::util::detail::zip_iterator_base<hpx::tuple<Ts_\ldots>, zip_iterator<Ts_\ldots>>

Public Functions

zip_iterator ()
zip_iterator (Ts const\&\ldots vs)
zip_iterator (hpx::tuple<Ts_\ldots> \&\&t)
zip_iterator (zip_iterator const \&other)
zip_iterator (zip_iterator \&\&other)
zip_iterator \& operator= (zip_iterator const \&other)
zip_iterator \& \& operator= (zip_iterator \&\&other)

template<\typename Ts_>
std::enable_if<\typename \ldots> zip_iterator::iterator_tuple_type\&, typename zip_iterator<Ts_\ldots>::iterator_tuple_type\&>::type
operator= (zip_iterator const \&other)
operator= (zip_iterator \&\&other)

Private Types

typedef detail::zip_iterator_base<hpx::tuple<Ts_\ldots>, zip_iterator<Ts_\ldots>> base_type

template<\typename Ts_>
class zip_iterator<hpx::tuple<Ts_\ldots>> : public hpx::util::detail::zip_iterator_base<hpx::tuple<Ts_\ldots>, zip_iterator<Ts_\ldots>>
Public Functions

```cpp
zip_iterator()
zip_iterator(Ts const&... vs)
zip_iterator(hpx::tuple<Ts...>&& t)
zip_iterator(zip_iterator const &other)
zip_iterator(zip_iterator &&other)
zip_iterator &operator=(zip_iterator const &other)
zip_iterator &operator=(zip_iterator &&other)
```

```cpp
template<typename ...Ts_>
std::enable_if<std::is_assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iterator_tuple_type>::value, zip_iterator&>::type operator=(zip_iterator<Ts_...> const &other)
```

```cpp
template<typename ...Ts_>
std::enable_if<std::is_assignable<typename zip_iterator::iterator_tuple_type&, typename zip_iterator<Ts_...>::iterator_tuple_type>::value, zip_iterator&>::type operator=(zip_iterator<Ts_...> &&other)
```

Private Types

```cpp
template<>
using base_type = detail::zip_iterator_base<hpx::tuple<Ts...>, zip_iterator<hpx::tuple<Ts...>>>
```

```cpp
namespace hpx

namespace traits

Typedefs

```cpp
template<
typename Iter>
using is_iterator_t = typename is_iterator<Iter>::type
```

```cpp
template<
typename Iter>
using is_output_iterator_t = typename is_output_iterator<Iter>::type
```

```cpp
template<
typename Iter>
using is_input_iterator_t = typename is_input_iterator<Iter>::type
```

```cpp
template<
typename Iter>
using is_forward_iterator_t = typename is_forward_iterator<Iter>::type
```

```cpp
template<
typename Iter>
using is_bidirectional_iterator_t = typename is_bidirectional_iterator<Iter>::type
```

```cpp
template<
typename Iter>
using is_random_access_iterator_t = typename is_random_access_iterator<Iter>::type
```

```cpp
template<
typename Iter>
using is_segmented_iterator_t = typename is_segmented_iterator<Iter>::type
```

using is_segmented_local_iterator_t = typename is_segmented_local_iterator<Iter>::type

template<typename Iter>
using is_zip_iterator_t = typename is_zip_iterator<Iter>::type

namespace hpx

namespace traits

Variables

template<typename Iter> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_iterator_v = is_iterator<Iter>::value

template<typename Iter, typename Category> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::has_category_v = has_category<Iter, Category>::value

template<typename Iter> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_output_iterator_v = is_output_iterator<Iter>::value

template<typename Iter> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_input_iterator_v = is_input_iterator<Iter>::value

template<typename Iter> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_forward_iterator_v = is_forward_iterator<Iter>::value

template<typename Iter> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_bidirectional_iterator_v = is_bidirectional_iterator<Iter>::value

template<typename Iter> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_random_access_iterator_v = is_random_access_iterator<Iter>::value

template<typename Iter> HPXINLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_segmented_iterator_v = is_segmented_iterator<Iter>::value

namespace hpx

namespace traits

Typedefs

template<typename T>
using range_iterator_t = typename range_iterator<T>::type

template<typename T>
using range_sentinel_t = typename range_sentinel<T>::type
Variables

template<
typename T>
HPX_INLINE_CONSTEXPR_VARIABLE
bool
hpx::traits::is_range_v = is_range<T>::value

template<
typename R>
struct
range_traits
<R, true> : public
std::iterator_traits<
util::detail::iterator<R>::type>

Public Types

template<>
using
iterator_type = typename
util::detail::iterator<R>::type

template<>
using
sentinel_type = typename
util::detail::sentinel<R>::type

namespace hpx

namespace traits

Variables

template<
typename Sent, typename Iter>
HPX_INLINE_CONSTEXPR_VARIABLE
bool
hpx::traits::is_sentinel_for_v = is_sentinel_for<Sent, Iter>::value

template<
typename Sent, typename Iter>
HPX_INLINE_CONSTEXPR_VARIABLE
bool
hpx::traits::disable_sized_sentinel_for = false

template<
typename Sent, typename Iter>
HPX_INLINE_CONSTEXPR_VARIABLE
bool
hpx::traits::is_sized_sentinel_for_v = is_sized_sentinel_for<Sent, Iter>::value

Functions

template<
typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool
operator== (Iter it, sentinel<ValueType> s)

template<
typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool
operator== (sentinel<ValueType> s, Iter it)

template<
typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool
operator!= (Iter it, sentinel<ValueType> s)

template<
typename Iter, typename ValueType, typename Enable = std::enable_if_t<hpx::traits::is_forward_iterator<Iter>::value>
bool
operator!= (sentinel<ValueType> s, Iter it)

template<
typename ValueType>
struct
sentinel

Public Functions

sentinel (ValueType stop_value)

ValueType get_stop () const
Private Members

ValueType stop
template<typename Value>
struct iterator

Public Types

template<>
using difference_type = std::ptrdiff_t
template<>
using value_type = Value
template<>
using iterator_category = std::forward_iterator_tag
template<>
using pointer = Value const*
template<>
using reference = Value const&

Public Functions

iterator (Value initialState)
virtual Value operator* () const
virtual Value operator-> () const
iterator &operator++ ()
iterator operator++ (int)
iterator &operator-- ()
iterator operator-- (int)
virtual Value operator[] (difference_type n) const
iterator &operator+= (difference_type n)
iterator operator+ (difference_type n) const
iterator &operator-= (difference_type n)
iterator operator- (difference_type n) const
bool operator== (const iterator &that) const
bool operator!= (const iterator &that) const
bool operator< (const iterator &that) const
bool operator<= (const iterator &that) const
bool operator> (const iterator &that) const
bool operator>= (const iterator &that) const
bool operator>= (const iterator &that) const

Protected Attributes

Value state

itt_notify

The contents of this module can be included with the header hpx/modules/itt_notify.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/itt_notify.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

HPX_ITT_SYNC_CREATE (obj, type, name)
HPX_ITT_SYNC_RENAME (obj, name)
HPX_ITT_SYNC_PREPARE (obj)
HPX_ITT_SYNC_CANCEL (obj)
HPX_ITT_SYNC_ACQUIRED (obj)
HPX_ITT_SYNC_RELEASING (obj)
HPX_ITT_SYNC_RELEASED (obj)
HPX_ITT_SYNC_DESTROY (obj)
HPX_ITT_STACK_CREATE (ctx)
HPX_ITT_STACK_CALLEE_ENTER (ctx)
HPX_ITT_STACK_CALLEE_LEAVE (ctx)
HPX_ITT_STACK_DESTROY (ctx)
HPX_ITT_FRAME_BEGIN (frame, id)
HPX_ITT_FRAME_END (frame, id)
HPX_ITT_MARK_CREATE (mark, name)
HPX_ITT_MARK_OFF (mark)
HPX_ITT_MARK (mark, parameter)
HPX_ITT_THREAD_SET_NAME (name)
HPX_ITT_THREAD_IGNORE ()
HPX_ITT_TASK_BEGIN (domain, name)
HPX_ITT_TASK_BEGIN_ID (domain, id, name)
HPX_ITT_TASK_END (domain)
HPX_ITT_DOMAIN_CREATE (name)
HPX_ITT_STRING_HANDLE_CREATE (name)
HPX Documentation, master

HPX_ITT_MAKE_ID (addr, extra)
HPX_ITT_ID_CREATE (domain, id)
HPX_ITT_ID_DESTROY (id)
HPX_ITT_HEAP_FUNCTION_CREATE (name, domain)
HPX_ITT_HEAP_ALLOCATE_BEGIN (f, size, initialized)
HPX_ITT_HEAP_ALLOCATE_END (f, addr, size, initialized)
HPX_ITT_HEAP_FREE_BEGIN (f, addr)
HPX_ITT_HEAP_FREE_END (f, addr)
HPX_ITT_HEAP_REALLOCATE_BEGIN (f, addr, new_size, initialized)
HPX_ITT_HEAP_REALLOCATE_END (f, addr, new_addr, new_size, initialized)
HPX_ITT_HEAP_INTERNAL_ACCESS_BEGIN ()
HPX_ITT_HEAP_INTERNAL_ACCESS_END ()
HPX_ITT_COUNTER_CREATE (name, domain)
HPX_ITT_COUNTER_CREATE_TYPED (name, domain, type)
HPX_ITT_COUNTER_SET_VALUE (id, value_ptr)
HPX_ITT_COUNTER_DESTROY (id)
HPX_ITT_METADATA_ADD (domain, id, key, data)

Typedefs

using __itt_heap_function = void*

Functions

constexpr void itt_sync_create (void*, const char*, const char*)
constexpr void itt_sync_rename (void*, const char*)
constexpr void itt_sync_prepare (void*)
constexpr void itt_sync_acquired (void*)
constexpr void itt_sync_cancel (void*)
constexpr void itt_sync_releasing (void*)
constexpr void itt_sync_released (void*)
constexpr void itt_sync_destroy (void*)
constexpr __itt_caller *itt_stack_create ()
constexpr void itt_stack_enter (__itt_caller*)
constexpr void itt_stack_leave (__itt_caller*)
constexpr void itt_stack_destroy (__itt_caller*)
constexpr void itt_frame_begin (__itt_domain const*, __itt_id*)
constexpr void itt_frame_end (__itt_domain const*, __itt_id*)
constexpr int itt_mark_create (char const *)
constexpr void itt_mark_off (int)
constexpr void itt_mark (int, char const *)
constexpr void itt_thread_set_name (char const *)
constexpr void itt_thread_ignore ()
constexpr void itt_task_begin (__itt_domain const*, __itt_string_handle*)
constexpr void itt_task_begin (__itt_domain const*, __itt_id*, __itt_string_handle*)
constexpr void itt_task_end (__itt_domain const*)
constexpr __itt_domain * itt_domain_create (char const *)
constexpr __itt_string_handle * itt_string_handle_create (char const *)
constexpr __itt_id * itt_make_id (void*, unsigned long)
constexpr void itt_id_create (__itt_domain const*, __itt_id*)
constexpr void itt_id_destroy (__itt_id*)
constexpr __itt_heap_function itt_heap_function_create (const char*, const char*)
constexpr void itt_heap_allocate_begin (__itt_heap_function, std::size_t, int)
constexpr void itt_heap_allocate_end (__itt_heap_function, void**, std::size_t, int)
constexpr void itt_heap_free_begin (__itt_heap_function, void*)
constexpr void itt_heap_free_end (__itt_heap_function, void*)
constexpr void itt_heap_reallocate_begin (__itt_heap_function, void*, std::size_t, int)
constexpr void itt_heap_reallocate_end (__itt_heap_function, void*, void**, std::size_t, int)
constexpr void itt_heap_internal_access_begin ()
constexpr void itt_heap_internal_access_end ()
constexpr __itt_counter * itt_counter_create (char const*, char const*)
constexpr __itt_counter * itt_counter_create_typed (char const*, char const*, int)
constexpr void itt_counter_destroy (__itt_counter*)
constexpr void itt_counter_set_value (__itt_counter*, void*)
constexpr int itt_event_create (char const*, int)
constexpr int itt_event_start (int)
constexpr int itt_event_end (int)
constexpr void itt_metadata_add (__itt_domain*, __itt_id*, __itt_string_handle*, std::uint64_t const&)
constexpr void itt_metadata_add (__itt_domain*, __itt_id*, __itt_string_handle*, double const&)
constexpr void itt_metadata_add (__itt_domain*, __itt_id*, __itt_string_handle*, char const*)
constexpr void itt_metadata_add (__itt_domain*, __itt_id*, __itt_string_handle*, void const*)

namespace hpx
namespace util

namespace itt

Functions

constexpr void event_tick(event const&)
struct caller_context

Public Functions

constexpr caller_context(stack_context&)
~caller_context()

struct counter

Public Functions

constexpr counter(char const*, char const*)
~counter()

struct domain
Subclassed by hpx::util::itt::thread_domain

Public Functions

HPX_NON_COPYABLE(domain)
constexpr domain(char const*)
domain()

struct event

Public Functions

constexpr event(char const*)

struct frame_context

Public Functions

```cpp
constexpr frame_context (domain const&, id* = nullptr)
~frame_context()
```

struct heap_allocate

Public Functions

```cpp
template<typename T>
constexpr heap_allocate (heap_function&, T**, std::size_t, int)
~heap_allocate()
```

struct heap_free

Public Functions

```cpp
constexpr heap_free (heap_function&, void*)
~heap_free()
```

struct heap_function

Public Functions

```cpp
constexpr heap_function (char const*, char const*)
~heap_function()
```

struct heap_internal_access

Public Functions

```cpp
heap_internal_access()
~heap_internal_access()
```

struct id

Public Functions

```cpp
constexpr id (domain const&, void*, unsigned long = 0)
~id()
```

struct mark_context
Public Functions

```cpp
constexpr mark_context (char const*)
~mark_context ()
```

```cpp
struct mark_event

Public Functions

```cpp
constexpr mark_event (event const&)
~mark_event ()
```

```cpp
struct stack_context

Public Functions

```cpp
stack_context ()
~stack_context ()
```

```cpp
struct string_handle

Public Functions

```cpp
constexpr string_handle (char const* = nullptr)
```

```cpp
struct task

Public Functions

```cpp
constexpr task (domain const&, string_handle const& , std::uint64_t) = default
constexpr task (domain const&, string_handle const& ) = default
~task ()
```

```cpp
struct thread_domain : public hpx::util::itt::domain

Public Functions

```cpp
HPX_NON_COPYABLE (thread_domain)
thread_domain ()
```

```cpp
struct undo_frame_context
Public Functions

```cpp
constexpr undo_frame_context (frame_context const&)  
~undo_frame_context ()
```

Public Functions

```cpp
constexpr undo_mark_context (mark_context const&)  
~undo_mark_context ()
```

lcos_local

The contents of this module can be included with the header hpx/modules/lcos_local.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/lcos_local.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

```cpp
namespace lcos

namespace local

struct and_gate : public hpx::lcos::local::base_and_gate<no_mutex>
```

Public Functions

```cpp
and_gate (std::size_t count = 0)  
and_gate (and_gate &&rhs)  
and_gate &operator=(and_gate &&rhs)
```

```cpp
template<typename Lock>  
future<void> get_future (Lock &l, std::size_t count = std::size_t(-1), std::size_t *generation_value = nullptr, error_code &ec = hpx::throws)
```

```cpp
template<typename Lock>  
shared_future<void> get_shared_future (Lock &l, std::size_t count = std::size_t(-1), 
std::size_t *generation_value = nullptr, error_code &ec = hpx::throws)
```

```cpp
bool set (std::size_t which, Lock l, error_code &ec = throws)
```

```cpp
template<typename Lock>  
void synchronize (std::size_t generation_value, Lock &l, char const *function_name = 
"and_gate::synchronize", error_code &ec = throws)
```
Private Types

typedef base_and_gate<no_mutex> base_type

template<typename Mutex = lcos::local::spinlock>
struct base_and_gate

Public Functions

base_and_gate (std::size_t count = 0)
This constructor initializes the base_and_gate object from the number of participants to synchronize the control flow with.

base_and_gate (base_and_gate &&rhs)
base_and_gate & operator= (base_and_gate &&rhs)

future<void> get_future (std::size_t count = std::size_t(-1), std::size_t *generation_value = nullptr, error_code &ec = hpx::throws)

shared_future<void> get_shared_future (std::size_t count = std::size_t(-1), std::size_t *generation_value = nullptr, error_code &ec = hpx::throws)

bool set (std::size_t which, error_code &ec = throws)

void synchronize (std::size_t generation_value, char const *function_name = "base_and_gate<>::synchronize", error_code &ec = throws)
Wait for the generational counter to reach the requested stage.

std::size_t next_generation ()

std::size_t generation () const

Protected Types

typedef Mutex mutex_type

Protected Functions

bool trigger_conditions (error_code &ec = throws)

template<typename OuterLock>
future<void> get_future (OuterLock &outer_lock, std::size_t count = std::size_t(-1),
std::size_t *generation_value = nullptr, error_code &ec = hpx::throws)
get a future allowing to wait for the gate to fire

template<typename OuterLock>
shared_future<void> get_shared_future (OuterLock &outer_lock, std::size_t count = std::size_t(-1),
std::size_t *generation_value = nullptr, error_code &ec = hpx::throws)
get a shared future allowing to wait for the gate to fire

template<typename OuterLock>
bool **set**(std::size_t which, OuterLock outer_lock, error_code &ec = throws)
Set the data which has to go into the segment which.

bool **test_condition**(std::size_t generation_value)

template<typename Lock>
void **synchronize**(std::size_t generation_value, Lock &l, char const *function_name = "base_and_gate<>\::\synchronize", error_code &ec = throws)

template<typename OuterLock, typename Lock>
void **init_locked**(OuterLock &outer_lock, Lock &l, std::size_t count, error_code &ec = throws)

**Private Types**

typedef std::list<conditional_trigger*> **condition_list_type**

**Private Members**

mutex_type mtx_
boost::dynamic_bitset received_segments_
lcos::local::promise<void> promise_
std::size_t generation_
condition_list_type conditions_

struct manage_condition

**Public Functions**

template<>
manage_condition(base_and_gate &gate, conditional_trigger &cond)

template<>
~manage_condition()

template<typename Condition>
future<void> **get_future**(Condition &&func, error_code &ec = hpx::throws)

**Public Members**

template<>
base_and_gate &this_

template<>
condition_list_type::iterator it_

namespace hpx

namespace lcos
namespace local

template<typename T>
class channel

Public Types

template<>
using value_type = T

Public Functions

channel()

Private Types

template<>
using base_type = detail::channel_base<T>

Friends

friend hpx::lcos::local::channel_iterator< T >
friend hpx::lcos::local::receive_channel< T >
friend hpx::lcos::local::send_channel< T >

template<>
class channel<void> : protected hpx::lcos::local::detail::channel_base<void>

Public Types

template<>
using value_type = void

Public Functions

channel()

Private Types

template<>
using base_type = detail::channel_base<void>
Friends

friend hpx::lcos::local::channel_iterator< void >
friend hpx::lcos::local::receive_channel< void >
friend hpx::lcos::local::send_channel< void >

template<typename T>
class channel_async_iterator : public hpx::util::iterator_facade<channel_async_iterator<T>, hpx::future<T>, std::input_iterator_tag, hpx::future<T>>

Public Functions

channel_async_iterator()
channel_async_iterator(detail::channel_base<T> const *c)

Private Types

template<>
using base_type = hpx::util::iterator_facade<channel_async_iterator<T>, hpx::future<T>, std::input_iterator_tag, hpx::future<T>>

Private Functions

std::pair<hpx::future<T>, bool> get_checked() const
bool equal(channel_async_iterator const &rhs) const
void increment()
base_type::reference dereference() const

Private Members

hpx::intrusive_ptr<detail::channel_impl_base<T>> channel_
std::pair<hpx::future<T>, bool> data_

Friends

friend hpx::lcos::local::hpx::util::iterator_core_access

template<typename T>
class channel_iterator : public hpx::util::iterator_facade<channel_iterator<T>, T const, std::input_iterator_tag, hpx::future<T>>
Public Functions

channel_iterator()

channel_iterator(detail::channel_base<T> const *c)

channel_iterator(receive_channel<T> const *c)

Private Types

template<>
using base_type = hpx::util::iterator_facade<channel_iterator<T>, T const, std::input_iterator_tag>

Private Functions

std::pair<T, bool> get_checked() const

bool equal(channel_iterator const &rhs) const

void increment()

base_type::reference dereference() const

Private Members

hpx::intrusive_ptr<detail::channel_impl_base<T>> channel_

std::pair<T, bool> data_

Friends

friend hpx::lcos::local::hpx::util::iterator_core_access

template<>
class channel_iterator<void> : public hpx::util::iterator_facade<channel_iterator<void>, util::unused_type>

Public Functions

channel_iterator()

channel_iterator(detail::channel_base<void> const *c)

channel_iterator(receive_channel<void> const *c)
Private Types

template<>  
using base_type = hpx::util::iterator_facade<channel_iterator<void>, util::unused_type const, std::input_iterator_tag>

Private Functions

bool get_checked()  
bool equal(channel_iterator const & rhs) const  
void increment()  
base_type::reference dereference() const

Private Members

hpx::intrusive_ptr<detail::channel_impl_base<util::unused_type> > channel_  
bool data_

Friends

friend hpx::lcos::local::hpx::util::iterator_core_access

template<typename T>  
class one_element_channel

Public Types

template<>  
using value_type = T

Public Functions

one_element_channel()  

Private Types

template<>  
using base_type = detail::channel_base<T>
Friends

friend hpx::lcos::local::channel_iterator< T >
friend hpx::lcos::local::receive_channel< T >
friend hpx::lcos::local::send_channel< T >

template<>
class one_element_channel< void > : protected hpx::lcos::local::detail::channel_base< void >

Public Types

template<>
using value_type = void

Public Functions

one_element_channel()

Private Types

template<>
using base_type = detail::channel_base< void >

Friends

friend hpx::lcos::local::channel_iterator< void >
friend hpx::lcos::local::receive_channel< void >
friend hpx::lcos::local::send_channel< void >

template<typename T>
class receive_channel

Public Functions

receive_channel(channel< T > const & c)
receive_channel(one_element_channel< T > const & c)

Private Types

template<>
using base_type = detail::channel_base< T >
Friends

friend hpx::lcos::local::channel_iterator< T >
friend hpx::lcos::local::send_channel< T >

template<>
class receive_channel<void> : protected hpx::lcos::local::detail::channel_base<void>

Public Functions

receive_channel(channel<void> const &c)
receive_channel(one_element_channel<void> const &c)

Private Types

template<>
using base_type = detail::channel_base<void>

Friends

friend hpx::lcos::local::channel_iterator< void >
friend hpx::lcos::local::send_channel< void >

template<typename T>
class send_channel

Public Functions

send_channel(channel<T> const &c)
send_channel(one_element_channel<T> const &c)

Private Types

template<>
using base_type = detail::channel_base<T>

template<>
class send_channel<void> : private hpx::lcos::local::detail::channel_base<void>
Public Functions

send_channel(channel<void> const &c)

send_channel(one_element_channel<void> const &c)

Private Types

template<>
using base_type = detail::channel_base<void>

namespace hpx

namespace lcos

namespace local

Functions

void run_guarded(guard &guard, detail::guard_function task)
   Conceptually, a guard acts like a mutex on an asynchronous task. The mutex is locked before the
task runs, and unlocked afterwards.

template<typename F, typename ... Args>
void run_guarded(guard &guard, F &&f, Args&&... args)

void run_guarded(guard_set &guards, detail::guard_function task)
   Conceptually, a guard_set acts like a set of mutexes on an asynchronous task. The mutexes are
locked before the task runs, and unlocked afterwards.

template<typename F, typename ... Args>
void run_guarded(guard_set &guards, F &&f, Args&&... args)

class guard : public hpx::lcos::local::detail::debug_object

Public Functions

guard()

~guard()

Public Members

detail::guard_atomic task

class guard_set : public hpx::lcos::local::detail::debug_object
Public Functions

`guard_set()`

`~guard_set()`

`std::shared_ptr<guard> get(size_t i)`

`void add(const &guard_ptr)`

`std::size_t size()`

Private Functions

`void sort()`

Private Members

`std::vector<std::shared_ptr<guard>> guards`

`bool sorted`

Friends

`void run_guarded(guard_set &guards, detail::guard_function task)`

Conceptually, a `guard_set` acts like a set of mutexes on an asynchronous task. The mutexes are locked before the task runs, and unlocked afterwards.

namespace hpx

namespace lcos

namespace local

struct conditional_trigger

Public Functions

`conditional_trigger()`

`conditional_trigger(conditional_trigger &&rhs)`

`conditional_trigger &operator= (conditional_trigger &&rhs)`

`template<typename Condition>`

`future<void> get_future(Condition &&func, error_code &ec = hpx::throws)`

get a future allowing to wait for the trigger to fire

`void reset()`

`bool set (error_code &ec = throws)`

Trigger this object.
Private Members

$lcos::local::promise<void>$\ promise_
$util::function_nonser<bool>() > cond_

namespace hpx

namespace lcos

namespace local

template<typename T, typename Mutex = lcos::local::spinlock>
struct receive_buffer

Public Functions

receive_buffer()
receive_buffer(receive_buffer&& other)
~receive_buffer()
receive_buffer& operator=(receive_buffer&& other)
$hpx::future<T>$ receive($std::size_t$ step)
bool try_receive($std::size_t$ step, $hpx::future<T>$ *f = nullptr)
template<typename Lock = hpx::lcos::local::no_mutex>
void store_received($std::size_t$ step, T&& val, Lock* lock = nullptr)
bool empty() const
$std::size_t$ cancel_waiting($std::exception_ptr$ const &e, bool force_delete_entries = false)

Protected Types

typedef Mutex mutex_type

typedef $hpx::lcos::local::promise<T>$ buffer.promise_type

typedef $std::map<std::size_t, std::shared_ptr<entry_data>>$ buffer.map_type

typedef buffer.map_type::iterator iterator
**Protected Functions**

*iterator get_buffer_entry (std::size_t step)*

**Private Members**

*mutex_type mtx_*

*buffer_map_type buffer_map_*

*struct entry_data*

**Public Functions**

*template<> HPX_NON_COPYABLE (entry_data)*

*template<> entry_data ()*

*template<> hpx::future<T> get_future ()*

*template<typename Val> void set_value (Val &&val)*

*template<> bool cancel (std::exception_ptr const &e)*

**Public Members**

*template<> buffer_promise_type promise_*

*template<> bool can_be_deleted_*

*template<> bool value_set_*

*struct erase_on_exit*

**Public Functions**

*template<> erase_on_exit (buffer_map_type &buffer_map, iterator it)*

*template<> ~erase_on_exit ()*
Public Members

template<>
buffer_map_type &buffer_map_

template<>
iterator it_

template<
type name Mutex>
struct receive_buffer<void, Mutex>

Public Functions

receive_buffer()
receive_buffer (receive_buffer &&other)
~receive_buffer()
receive_buffer &operator= (receive_buffer &&other)

hpx::future<void> receive (std::size_t step)

bool try_receive (std::size_t step, hpx::future<void> *f = nullptr)

template<
type name Lock = hpx::lcos::local::no_mutex>
void store_received (std::size_t step, Lock *lock = nullptr)

bool empty () const

std::size_t cancel_waiting (std::exception_ptr const &e, bool force_delete_entries = false)

Protected Types

typedef Mutex mutex_type

typedef hpx::lcos::local::promise<void> buffer_promise_type

typedef std::map<std::size_t, std::shared_ptr<entry_data>> buffer_map_type

typedef buffer_map_type::iterator iterator

Protected Functions

iterator get_buffer_entry (std::size_t step)
Private Members

mutex_type mtx_
buffer_map_type buffer_map_

template<> struct entry_data

Public Functions

template<>
HPX_NON_COPYABLE(entry_data)

template<> entry_data()

template<> hpx::future<void> get_future()

template<> void set_value()

template<> bool cancel(std::exception_ptr const & e)

Public Members

template<>
buffer_promise_type promise_

template<> bool can_be_deleted_

template<> bool value_set_

template<>
struct erase_on_exit

Public Functions

template<>
erase_on_exit(buffer_map_type & buffer_map, iterator it)

template<>
~erase_on_exit()
Public Members

template<>
buffer_map_type &buffer_map_

template<>
iterator it_

namespace hpx

namespace lcos

namespace local

namespace hpx

namespace lcos

namespace local


Public Functions

base_trigger()

base_trigger (base_trigger &rhs)

base_trigger &operator= (base_trigger &rhs)

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

bool set (error_code &ec = throws)

Trigger this object.

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.

std::size_t next_generation ()

std::size_t generation () const

Protected Types

typedef Mutex mutex_type
Protected Functions

bool trigger_conditions (error_code &ec = throws)

template< typename Lock>
void synchronize (std::size_t generation_value, Lock &l, char const *function_name = "trigger::synchronize", error_code &ec = throws)

Private Types

typedef std::list<conditional_trigger*> condition_list_type

Private Functions

bool test_condition (std::size_t generation_value)

Private Members

mutex_type mtx_
lcos::local::promise<void> promise_
std::size_t generation_
condition_list_type conditions_

struct manage_condition

Public Functions

template<>
manage_condition (base_trigger &gate, conditional_trigger &cond)

template<>
~manage_condition ()

template< typename Condition>
future<void> get_future (Condition &&func, error_code &ec = hpx::throws)

Public Members

template<>
base_trigger &this_

template<>
condition_list_type::iterator it_

struct trigger : public hpx::lcos::local::base_trigger<no_mutex>
Public Functions

```cpp
trigger()

trigger(trigger &&rhs)

trigger &operator=(trigger &&rhs)

template<typename Lock>
void synchronize(std::size_t generation_value, Lock &l, char const *function_name = 
"trigger::synchronize", error_code &ec = throws)
```

Private Types

```cpp
typedef base_trigger<no_mutex> base_type
```

logging

The contents of this module can be included with the header hpx/modules/logging.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/logging.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace util

namespace logging

Enums

```cpp
enum level

Handling levels - classes that can hold and/or deal with levels.

- filters and level holders

By default we have these levels:

- debug (smallest level),
- info,
- warning ,
- error ,
- fatal (highest level)
```

Depending on which level is enabled for your application, some messages will reach the log: those messages having at least that level. For instance, if info level is enabled, all logged messages will reach the log. If warning level is enabled, all messages are logged, but the warnings. If debug level is enabled, messages that have levels debug, error, fatal will be logged.

Values:

```cpp
disable_all = static_cast<unsigned int>(-1)
```
enable_all = 0
debug = 1000
info = 2000
warning = 3000
error = 4000
fatal = 5000
always = 6000

**Functions**

```cpp
void format_value (std::ostream &os, boost::string_ref spec, level value)
```

Include this file when you’re using the logging lib, but don’t necessarily want to use formatters and destinations. If you want to use formatters and destinations, then you can include this one instead:

```cpp
#include <hpx/logging/format.hpp>
```

```cpp
namespace hpx
```

```cpp
namespace util
```

```cpp
namespace logging
```

```cpp
namespace destination
```

Destination is a manipulator. It contains a place where the message, after being formatted, is to be written to.

Some viable destinations are: the console, a file, a socket, etc.

```cpp
struct manipulator
```

```cpp
#include <manipulator.hpp> What to use as base class, for your destination classes.
```

Subclassed by `hpx::util::logging::destination::cerr`, `hpx::util::logging::destination::cout`, `hpx::util::logging::destination::dbg_window`, `hpx::util::logging::destination::file`, `hpx::util::logging::destination::stream`

**Public Functions**

```cpp
virtual void operator() (message const&) = 0
```

```cpp
virtual void configure (std::string const&)
```

Override this if you want to allow configuration through scripting.

That is, this allows configuration of your manipulator at run-time.

```cpp
virtual ~manipulator()
```
Protected Functions

manipulator()

namespace formatter

Formatter is a manipulator. It allows you to format the message before writing it to the destination(s).

Examples of formatters are: prepend the time, prepend high-precision time, prepend the index of the message, etc.

struct manipulator

#include <manipulator.hpp> What to use as base class, for your formatter classes.

Subclassed by hpx::util::logging::formatter::high_precision_time, hpx::util::logging::formatter::idx, hpx::util::logging::formatter::thread_id

Public Functions

virtual void operator() (std::ostream&) const = 0

virtual void configure (std::string const&)  
Override this if you want to allow configuration through scripting.
That is, this allows configuration of your manipulator at run-time.

virtual ~manipulator()

Protected Functions

manipulator()

Friends

void format_value (std::ostream &os, boost::string_ref, manipulator const &value)

namespace hpx

namespace util

namespace logging

class message

#include <message.hpp> Optimizes the formatting for prepending and/or appending strings to the original message.

It keeps all the modified message in one string. Useful if some formatter needs to access the whole string at once.

reserve() - the size that is reserved for prepending (similar to string::reserve function)

Note: as strings are prepended, reserve() shrinks.

1004 Chapter 2. What’s so special about HPX?
Public Functions

message ()

message (std::stringstream msg)

Parameters
  • msg: - the message that is originally cached

message (message &other)

template<typename T>
message & operator<< (T &v)

template<typename ...Args>
message & format (boost::string_ref format_str, Args const &... args)

std::string const & full_string () const
  returns the full string

bool empty () const

Private Members

std::stringstream m_str

bool m_full_msg_computed

std::string m_full_msg

Friends

std::ostream & operator<< (std::ostream & os, message const & value)

namespace hpx

namespace util

namespace logging

namespace destination

namespace destination

Destination is a manipulator. It contains a place where the message, after being formatted, is to
be written to.

Some viable destinations are: the console, a file, a socket, etc.

struct cerr : public hpx::util::logging::destination::manipulator

#include <destinations.hpp> Writes the string to cerr.
Public Functions

~cerr()

Public Static Functions

static std::unique_ptr<cerr> make()

Protected Functions

cerr()

struct cout : public hpx::util::logging::destination::manipulator
#include <destinations.hpp> Writes the string to console.

Public Functions

~cout()

Public Static Functions

static std::unique_ptr<cout> make()

Protected Functions

cout()

struct dbg_window : public hpx::util::logging::destination::manipulator
#include <destinations.hpp> Writes the string to output debug window.
For non-Windows systems, this is the console.

Public Functions

~dbg_window()

Public Static Functions

static std::unique_ptr<dbg_window> make()
Protected Functions

`dbg_window()`

```cpp
struct file : public hpx::util::logging::destination::manipulator
#include <destinations.hpp> Writes the string to a file.
```

Public Functions

`~file()`

Public Static Functions

```cpp
static std::unique_ptr<file> make(const std::string& file_name, file_settings set = {})
constructs the file destination
```

Parameters
- `file_name`: name of the file
- `set`: [optional] file settings - see `file_settings` class, and dealing_with_flags

Protected Functions

```cpp
file(const string& file_name, file_settings set)
```

Protected Attributes

```cpp
std::string name
file_settings settings
```

```cpp
struct file_settings
#include <destinations.hpp> settings for when constructing a file class. To see how it’s used,
see dealing_with_flags.
```

Public Functions

```cpp
file_settings()
```

Public Members

```cpp
bool flush_each_time : 1
    if true (default), flushes after each write
bool initial_overwrite : 1
bool do_append : 1
std::ios_base::openmode extra_flags
    just in case you have some extra flags to pass, when opening the file
```
struct stream: public hpx::util::logging::destination::manipulator
#include <destinations.hpp> writes to stream.

Note: The stream must outlive this object! Or, clear() the stream, before the stream is deleted.

Public Functions

~stream()

void set_stream (std::ostream *stream_ptr)
    resets the stream. Further output will be written to this stream

void clear()
    clears the stream. Further output will be ignored

Public Static Functions

static std::unique_ptr<stream> make (std::ostream *stream_ptr)

Protected Functions

stream (std::ostream *stream_ptr)

Protected Attributes

std::ostream *ptr

namespace hpx

namespace util

namespace logging

namespace formatter

Formatter is a manipulator. It allows you to format the message before writing it to the destination(s)

Examples of formatters are: prepend the time, prepend high-precision time, prepend the index of the message, etc.

struct high_precision_time: public hpx::util::logging::formatter::manipulator
#include <formatters.hpp> Prefixes the message with a high-precision time (. You pass the format string at construction.

#include <hpx/logging/format/formatter/high_precision_time.hpp>
Internally, it uses \texttt{hpx::util::date\_time::microsec\_time\_clock}. So, our precision matches this class.

The format can contain escape sequences: $dd$ - day, 2 digits $MM$ - month, 2 digits $yy$ - year, 2 digits $yyyy$ - year, 4 digits $hh$ - hour, 2 digits $mm$ - minute, 2 digits $ss$ - second, 2 digits $mili$ - milliseconds $micro$ - microseconds (if the high precision clock allows; otherwise, it pads zeros) $nano$ - nanoseconds (if the high precision clock allows; otherwise, it pads zeros)

Example:

\begin{verbatim}
high_precision_time("$mm:$ss:$micro");
\end{verbatim}

**Parameters**

- \texttt{convert}  [optional] In case there needs to be a conversion between std::(w)string and the string that holds your logged message. See convert\_format.

**Public Functions**

\begin{verbatim}
~high_precision_time()
\end{verbatim}

**Public Static Functions**

\begin{verbatim}
static std::unique\_ptr<high\_precision\_time> make (std::string const & format)
\end{verbatim}

**Protected Functions**

\begin{verbatim}
high\_precision\_time (std::string const & format)
\end{verbatim}

**struct idx**  
\begin{verbatim}
public hpx::util::logging::formatter::manipulator
#include <formatters.hpp> prefixes each message with an index.
\end{verbatim}

Example:

\begin{verbatim}
L_ << "my message";
L_ << "my 2nd message";
\end{verbatim}

This will output something similar to:

\begin{verbatim}
[1] my message
[2] my 2nd message
\end{verbatim}
Public Functions

~idx()

Public Static Functions

static std::unique_ptr<idx> make()

Protected Functions

idx()

struct thread_id : public hpx::util::logging::formatter::manipulator
#include <formatters.hpp> Writes the thread_id to the log.

Parameters

• convert: [optional] In case there needs to be a conversion between std::(w)string and the string that holds your logged message. See convert_format.

Public Functions

~thread_id()

Public Static Functions

static std::unique_ptr<thread_id> make()

Protected Functions

thread_id()

namespace hpx

namespace util

namespace logging

namespace writer

struct named_write
#include <named_write.hpp> Composed of a named formatter and a named destinations. Thus, you can specify the formatting and destinations as strings.
Contains a very easy interface for using formatters and destinations:

- at construction, specify 2 params: the formatter string and the destinations string

Setting the formatters and destinations to write to is extremely simple:

```cpp
// Set the formatters (first param) and destinations (second step) in one step
ml()->writer().write("%time($hh:$mm:$ss.$mili) [%idx%] |\n", "cout file(out.txt) debug");

// set the formatter(s)
g_l()->writer.format("%time($hh:$mm:$ss.$mili) [%idx%] |\n");

// set the destination(s)
g_l()->writer.destination("cout file(out.txt) debug");
```

### Public Functions

**named_write()**

```cpp
void format (std::string const &format_str)
sets the format string: what should be before, and what after the original message, separated by "|"

Example: "[%idx%] \n" - this writes "[%idx%] " before the message, and "\n" after the message
If "|" is not present, the whole message is prepended to the message
```

```cpp
void destination (std::string const &destination_str)
sets the destinations string - where should logged messages be outputted
```

```cpp
void write (std::string const &format_str, std::string const &destination_str)
Specifies the formats and destinations in one step.
```

```cpp
void operator() (message const &msg) const
```

```cpp
template<typename Formatter>
void set_formatter (std::string const &name, Formatter fmt)
Replaces a formatter from the named formatter.

You can use this, for instance, when you want to share a formatter between multiple named writers.

```cpp
template<typename Formatter, typename ..Args>
void set_formatter (std::string const &name, Args&&... args)
```

```cpp
template<typename Destination>
void set_destination (std::string const &name, Destination dest)
Replaces a destination from the named destination.

You can use this, for instance, when you want to share a destination between multiple named writers.

```cpp
template<typename Destination, typename ..Args>
void set_destination (std::string const &name, Args&&... args)
```
Private Functions

void configure_formatter (std::string const &format)
void configure_destination (std::string const &format)

Private Members

detail::named_formatters m_format
detail::named_destinations m_destination
std::string m_format_str
std::string m_destination_str

Defines

LAGAS_(lvl)
LPT_(lvl)
LTIM_(lvl)
LPROGRESS_
LHPX_(lvl, cat)
LAPP_(lvl)
LDEB_
LTM_(lvl)
LRT_(lvl)
LOSH_(lvl)
LERR_(lvl)
LLCO_(lvl)
LPCS_(lvl)
LAS_(lvl)
LBT_(lvl)
LFATAL_
LAGAS_CONSOLE_(lvl)
LPT_CONSOLE_(lvl)
LTIM_CONSOLE_(lvl)
LHPX_CONSOLE_(lvl)
LAPP_CONSOLE_(lvl)
LDEB_CONSOLE_
LAGAS_ENABLED (lvl)
LPT_ENABLED (lvl)
LTIM_ENABLED (lvl)
LHPX_ENABLED (lvl)
LAPP_ENABLED (lvl)
LDEB_ENABLED

Functions

template<typename T>
bootstrap_logging const &operator<< (bootstrap_logging const &l, T&&)

Variables

constexpr bootstrap_logging lbt_
struct bootstrap_logging

Public Functions

constexpr bootstrap_logging()  
namespace hpx

Enums

enum logging_destination
  Values:
  destination_hpx = 0
  destination_timing = 1
  destination_agas = 2
  destination_parcel = 3
  destination_app = 4
  destination_debuglog = 5

memory

The contents of this module can be included with the header hpx/modules/memory.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/memory.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

template<typename T>
struct hash<hpx::memory::intrusive_ptr<T>>
Public Types

template<>
using result_type = std::size_t

Public Functions

result_type operator() (hpx::memory::intrusive_ptr<T> const &p) const

namespace hpx

namespace memory

Functions

template<typename T, typename U>
bool operator== (intrusive_ptr<T> const &a, intrusive_ptr<U> const &b)

template<typename T, typename U>
bool operator!= (intrusive_ptr<T> const &a, intrusive_ptr<U> const &b)

template<typename T, typename U>
bool operator== (intrusive_ptr<T> const &a, U *b)

template<typename T, typename U>
bool operator!= (intrusive_ptr<T> const &a, U *b)

template<typename T, typename U>
bool operator== (T *a, intrusive_ptr<U> const &b)

template<typename T, typename U>
bool operator!= (T *a, intrusive_ptr<U> const &b)

template<typename T>
bool operator== (intrusive_ptr<T> const &p, std::nullptr_t)

template<typename T>
bool operator== (std::nullptr_t, intrusive_ptr<T> const &p)

template<typename T>
bool operator!= (intrusive_ptr<T> const &p, std::nullptr_t)

template<typename T>
bool operator!= (std::nullptr_t, intrusive_ptr<T> const &p)

template<typename T>
bool operator< (intrusive_ptr<T> const &a, intrusive_ptr<T> const &b)

template<typename T>
void swap (intrusive_ptr<T> &lhs, intrusive_ptr<T> &rhs)

template<typename T>
T *get_pointer (intrusive_ptr<T> const &p)

template<typename T, typename U>
intrusive_ptr<T> static_pointer_cast (intrusive_ptr<U> const &p)

template<typename T, typename U>
intrusive_ptr<T> const_pointer_cast (intrusive_ptr<U> const &p)

template<typename T, typename U>
intrusive_ptr<T> dynamic_pointer_cast (intrusive_ptr<U> const &p)

template<typename T, typename U>
intrusive_ptr<T> static_pointer_cast (intrusive_ptr<U> &p)

template<typename T, typename U>
intrusive_ptr<T> const_pointer_cast (intrusive_ptr<U> &p)

template<typename T, typename U>
intrusive_ptr<T> dynamic_pointer_cast (intrusive_ptr<U> &p)

template<typename Y>
std::ostream &operator<< (std::ostream &os, intrusive_ptr<Y> const &p)

template<typename T>
class intrusive_ptr

Public Types

template<>
using element_type = T

Public Functions

constexpr intrusive_ptr ()

intrusive_ptr (T *, bool add_ref = true)

template<typename U, typename Enable = typename std::enable_if<
memory::detail::sp_convertible<U, T>::value>::type>
intrusive_ptr (intrusive_ptr<U> const &rhs)

intrusive_ptr (intrusive_ptr const &rhs)

~intrusive_ptr ()

template<typename U>
intrusive_ptr &operator= (intrusive_ptr<U> const &rhs)

constexpr intrusive_ptr (intrusive_ptr &rhs)

intrusive_ptr &operator= (intrusive_ptr &rhs)

template<typename U, typename Enable = typename std::enable_if<
memory::detail::sp_convertible<U, T>::value>::type>
constexpr intrusive_ptr (intrusive_ptr<U> &&rhs)

template<typename U>
intrusive_ptr &operator= (intrusive_ptr<U> &&rhs)

intrusive_ptr &operator= (intrusive_ptr const &rhs)

intrusive_ptr &operator= (T *rhs)
HPX Documentation, master

void reset ()
void reset (T *rhs)
void reset (T *rhs, bool add_ref)
constexpr T *get () const
constexpr T *detach ()
T &operator* () const
T *operator-> () const
constexpr operator bool () const
constexpr void swap (intrusive_ptr &rhs)

Private Types

template<>
using this_type = intrusive_ptr

Private Members

T *px = nullptr

Friends

friend hpx::memory::intrusive_ptr

namespace std

template<typename T>
struct hash<hpx::memory::intrusive_ptr<T>>

Public Types

template<>
using result_type = std::size_t

Public Functions

result_type operator() (hpx::memory::intrusive_ptr<T> const &p) const

namespace hpx

namespace serialization
Functions

```cpp
template<typename T>
void load (input_archive & ar, hpx::intrusive_ptr<T> & ptr, unsigned)

template<typename T>
void save (output_archive & ar, hpx::intrusive_ptr<T> const & ptr, unsigned)
```

**mpi_base**

The contents of this module can be included with the header `hpx/modules/mpi_base.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/mpi_base.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace util

struct mpi_environment

Public Static Functions

static bool check_mpi_environment (runtime_configuration const & cfg)
```

**pack_traversal**

The contents of this module can be included with the header `hpx/modules/pack_traversal.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/pack_traversal.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace util

Functions

```cpp
template<typename Mapper, typename... T><unspecified> hpx::util::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);

Return  The mapped element or in case the pack contains multiple elements, the pack is wrapped into a hpx::tuple.

Exceptions
• 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.

namespace hpx

namespace util

Functions

template<typename Visitor, typename ...T>
auto traverse_pack_async(Visitor &&visitor, 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:

struct my_async_visitor
{
    template <typename T>
    bool operator()(async_traverse_visit_tag, T&& element)
    {
        return true;
    }

    template <typename T, typename N>
    void operator()(async_traverse_detach_tag, T&& element, N&& next)
    {
    }

    template <typename T>
    void operator()(async_traverse_complete_tag, T&& pack)
    {
    }
};

See traverse_pack for a detailed description about the traversal behavior and capabilities.

Return  A hpx::intrusive_ptr that references an instance of the given visitor object.

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.

```cpp
template<typename Allocator, typename Visitor, typename ...T>
auto traverse_pack_async_allocator(Allocator const &alloc, Visitor &&visitor, 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:

```cpp
struct my_async_visitor
{
    template<typename T>
    bool operator()(async_traverse_visit_tag, T&& element)
    {
        return true;
    }

    template<typename T, typename N>
    void operator()(async_traverse_detach_tag, T&& element, N&& next)
    {
    }

    template<typename T>
    void operator()(async_traverse_complete_tag, T&& pack)
    {
    }
};
```

See `traverse_pack` for a detailed description about the traversal behavior and capabilities.

**Return** A `hpx::intrusive_ptr` that references an instance of the given visitor object.

**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.

• **alloc**: Allocator instance to use to create the traversal frame.
Functions

template<typename ...Args>
auto unwrap (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<hpx::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.

```cpp
// Single arguments
int i1 = 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.

Return 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.

Parameters

- args: the arguments that are unwrapped which may contain any arbitrary future or non future type.

Exceptions

- 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.

```
template<
  std::size_t Depth,
  typename ...Args>
 auto unwrap_n (Args&&... args)

An alternative 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)
    An alternative version of hpx::unwrap(), which unwarps 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)
    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.

    auto callable = hpx::unwrapping([](int left, int right) {
        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)
    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)
    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.

namespace util
Functions

template<typename ...Args>
auto unwrap (Args&&... args)

template<std::size_t Depth, typename ...Args>
auto unwrap_n (Args&&... args)

template<typename ...Args>
auto unwrap_all (Args&&... args)

template<typename T>
auto unwrapping (T &&callable)

template<std::size_t Depth, typename T>
auto unwrapping_n (T &&callable)

template<typename T>
auto unwrapping_all (T &&callable)

namespace functional

Public Types

template<>
using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>

Public Static Functions

static std::vector<NewType, NewAllocator> call (std::vector<OldType, OldAllocator> const &container)

namespace functional

Public Types

template<>
using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>
Public Static Functions

\texttt{static std::list<NewType, NewAllocator> call (std::list<OldType, OldAllocator> \ const &container)}

template<typename \texttt{NewType}, typename \texttt{OldType}, std::size_t \texttt{N}>
\texttt{struct pack_traversal_rebind_container<NewType, std::array<OldType, \texttt{N}>>}

Public Static Functions

\texttt{static std::array<NewType, \texttt{N}> call (std::array<OldType, \texttt{N}> \ const &)}

namespace \texttt{hpx}

namespace \texttt{traits}

template<typename \texttt{NewType}, typename \texttt{OldType}, std::size_t \texttt{N}>
\texttt{struct pack_traversal_rebind_container<NewType, std::array<OldType, \texttt{N}>>}

Public Static Functions

\texttt{static std::array<NewType, \texttt{N}> call (std::array<OldType, \texttt{N}> \ const \&)}

template<typename \texttt{NewType}, typename \texttt{OldType}, typename \texttt{OldAllocator}>
\texttt{struct pack_traversal_rebind_container<NewType, std::list<OldType, OldAllocator>>}

Public Types

template<>
\texttt{using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>}

Public Static Functions

\texttt{static std::list<NewType, NewAllocator> call (std::list<OldType, OldAllocator> \ const \&container)}

template<typename \texttt{NewType}, typename \texttt{OldType}, typename \texttt{OldAllocator}>
\texttt{struct pack_traversal_rebind_container<NewType, std::vector<OldType, OldAllocator>>}

Public Types

template<>
\texttt{using NewAllocator = typename std::allocator_traits<OldAllocator>::template rebind_alloc<NewType>
**Public Static Functions**

```cpp
static std::vector<NewType, NewAllocator> call (std::vector<OldType, OldAllocator> const & container)
```

**plugin**

The contents of this module can be included with the header `hpx/modules/plugin.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/plugin.hpp`, not the particular header in which the functionality you would like to use is defined. See *Public API* for a list of names that are part of the public *HPX* API.

**Defines**

- `HPX_PLUGIN_EXPORT_API`
- `HPX_PLUGIN_API`
- `HPX_PLUGIN_ARGUMENT_LIMIT`
- `HPX_PLUGIN_SYMBOLS_PREFIX_DYNAMIC`
- `HPX_PLUGIN_SYMBOLS_PREFIX`
- `HPX_PLUGIN_SYMBOLS_PREFIX_DYNAMIC_STR`
- `HPX_PLUGIN_SYMBOLS_PREFIX_STR`

```cpp
namespace hpx
{
    namespace util
    {
        namespace plugin
        {
            using shared_ptr = boost::shared_ptr<T>

            Defines
            - `HPX_HAS_DLOPEN`
        }
    }
}
```
Defines

HPX_PLUGIN_NAME_2 (name1, name2)
HPX_PLUGIN_NAME_3 (name, base, cname)
HPX_PLUGIN_LIST_NAME_ (prefix, name, base)
HPX_PLUGIN_EXPORTER_NAME_ (prefix, name, base, cname)
HPX_PLUGIN_EXPORTER_INSTANCE_NAME_ (prefix, name, base, cname)
HPX_PLUGIN_FORCE_LOAD_NAME_ (prefix, name, base)
HPX_PLUGIN_LIST_NAME (name, base)
HPX_PLUGIN_EXPORTER_NAME (name, base, cname)
HPX_PLUGIN_EXPORTER_INSTANCE_NAME (name, base, cname)
HPX_PLUGIN_FORCE_LOAD_NAME (name, base)
HPX_PLUGIN_LIST_NAME_DYNAMIC (name, base)
HPX_PLUGIN_EXPORTER_NAME_DYNAMIC (name, base, cname)
HPX_PLUGIN_EXPORTER_INSTANCE_NAME_DYNAMIC (name, base, cname)
HPX_PLUGIN_FORCE_LOAD_NAME_DYNAMIC (name, base)
HPX_PLUGIN_EXPORT_ (prefix, name, BaseType, ActualType, actualname, classname)
HPX_PLUGIN_EXPORT (name, BaseType, ActualType, actualname, classname)
HPX_PLUGIN_EXPORT_DYNAMIC (name, BaseType, ActualType, actualname, classname)
HPX_PLUGIN_EXPORT_LIST_ (prefix, name, classname)
HPX_PLUGIN_EXPORT_LIST (name, classname)
HPX_PLUGIN_EXPORT_LIST_DYNAMIC (name, classname)

namespace hpx

namespace util

namespace plugin

    template<class BasePlugin>
    struct plugin_factory : public hpx::util::plugin::detail::plugin_factory_item<BasePlugin, detail::plugin_factory_item_base, virtual_constructor<BasePlugin>::type>

Public Functions

    plugin_factory (dll &d, std::string const &basename)
Private Types

template<>
using base_type = detail::plugin_factory_item<BasePlugin, detail::plugin_factory_item_base, typename virtual_constructor<BasePlugin>::type>

template<class BasePlugin>
struct static_plugin_factory : public hpx::util::plugin::detail::static_plugin_factory_item<BasePlugin, detail::static_plugin_factory_item_base, virtual_constructor<BasePlugin>::type>

Public Functions

static_plugin_factory (get_plugins_list_type const &f)

Private Types

template<>
using base_type = detail::static_plugin_factory_item<BasePlugin, detail::static_plugin_factory_item_base, typename virtual_constructor<BasePlugin>::type>

namespace hpx

namespace util

namespace plugin

template<typename Wrapped, typename ...Parameters>
struct plugin_wrapper : public hpx::util::plugin::detail::dll_handle_holder, public Wrapped

Public Functions

plugin_wrapper (dll_handle dll, Parameters... parameters)

namespace hpx

namespace util

namespace plugin

Typedefs

using exported_plugins_type = std::map<std::string, hpx::any_nonser>
typedef exported_plugins_type*(HPX_PLUGIN_API* hpx::util::plugin::get_plugins_list_type)
typedef exported_plugins_type* HPX_PLUGIN_API hpx::util::plugin::get_plugins_list_np()
using dll_handle = shared_ptr<get_plugins_list_np>
template<typename BasePlugin>
struct virtual_constructor
Public Types

template<>
using type = hpx::util::pack<>

namespace hpx

namespace traits

    template<typename Plugin, typename Enable = void>
    struct plugin_config_data

Public Static Functions

    static char const *call()

prefix

The contents of this module can be included with the header hpx/modules/prefix.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/prefix.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

HPX_BASE_DIR_NAME
HPX_DEFAULT_INI_PATH
HPX_DEFAULT_INI_FILE
HPX_DEFAULT_COMPONENT_PATH

namespace hpx

namespace util

Functions

void set_hpx_prefix(const char *prefix)
char const *hpx_prefix()
std::string find_prefix(std::string const &library = "hpx")
std::string find_prefixes(std::string const &suffix, std::string const &library = "hpx")
std::string get_executable_filename (char const *argv0 = nullptr)
std::string get_executable_prefix (char const *argv0 = nullptr)
The contents of this module can be included with the header `hpx/modules/preprocessor.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/preprocessor.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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_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.

**Parameters**

- X: Token to be expanded twice

Example:

```c
#define MACRO(a, b, c) (a)(b)(c)
#define ARGS() (1, 2, 3)
HPX_PP_EXPAND(MACRO ARGS()) // expands to (1)(2)(3)
```

`HPX_PP_NARGS(...)`

Expands to the number of arguments passed in

**Example Usage:**

```c
HPX_PP_NARGS(hpx, pp, nargs)
HPX_PP_NARGS(hpx, pp)
HPX_PP_NARGS(hpx)
```

**Parameters**

- \ldots: The variadic number of arguments
Expands to:

```
3
2
1
```

**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

**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:**

**Parameters**

- `X`: Symbol to strip parens from

This produces the following output

```
no_parens
with_parens
```

**program_options**

The contents of this module can be included with the header `hpx/modules/program_options.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we *strongly* suggest only including the module header `hpx/modules/program_options.hpp`, not the particular header in which the functionality you would like to use is defined. See *Public API* for a list of names that are part of the public **HPX** API.

**namespace hpx**
namespace program_options

namespace command_line_style

Enums

enum style_t
    Various possible styles of options.

    There are “long” options, which start with “–” and “short”, which start with either “-” or “/”. Both kinds can be allowed or disallowed, see allow_long and allow_short. The allowed character for short options is also configurable.

    Option’s value can be specified in the same token as name (“–foo=bar”), or in the next token.

    It’s possible to introduce long options by the same character as short options, see allow_long_disguise.

    Finally, guessing (specifying only prefix of option) and case insensitive processing are supported.

Values:

allow_long = 1
    Allow “–long_name” style.

allow_short = allow_long << 1
    Allow “-<single character” style.

allow_dash_for_short = allow_short << 1
    Allow “-” in short options.

allow_slash_for_short = allow_dash_for_short << 1
    Allow “/” in short options.

long_allow_adjacent = allow_slash_for_short << 1
    Allow option parameter in the same token for long option, like in

        --foo=10

long_allow_next = long_allow_adjacent << 1
    Allow option parameter in the next token for long options.

short_allow_adjacent = long_allow_next << 1
    Allow option parameter in the same token for short options.

short_allow_next = short_allow_adjacent << 1
    Allow option parameter in the next token for short options.

allow_sticky = short_allow_next << 1
    Allow to merge several short options together, so that “–s -k” become “–sk”. All of the options but last should accept no parameter. For example, if “–s” accept a parameter, then “k” will be taken as parameter, not another short option. Dos-style short options cannot be sticky.

allow_guessing = allow_sticky << 1
    Allow abbreviated spellings for long options, if they unambiguously identify long option. No long option name should be prefix of other long option name if guessing is in effect.

long_case_insensitive = allow_guessing << 1
    Ignore the difference in case for long options.
short_case_insensitive = long_case_insensitive << 1
Ignore the difference in case for short options.

case_insensitive = (long_case_insensitive | short_case_insensitive)
Ignore the difference in case for all options.

allow_long_disguise = short_case_insensitive << 1
Allow long options with single option starting character, e.g. -foo=10

unix_style = (allow_short | short_allow_adjacent | short_allow_next | allow_long | long_allow_adjacent | long_allow_next | allow_sticky | allow_guessing | allow_dash_for_short)
The more-or-less traditional unix style.

default_style = unix_style
The default style.

namespace hpx

namespace program_options

Typedefs

using any = hpx::any

namespace hpx

namespace program_options

template<typename T>
using optional = hpx::util::optional<T>

namespace hpx

namespace program_options

class environment_iterator : public hpx::program_options::eof_iterator<environment_iterator, std::pair<std::string, std::string>>

Public Functions

environment_iterator(char **environment)

environment_iterator()

void get()

Private Members

char **m_environment

namespace hpx

namespace program_options

template<class Derived, class ValueType>
class eof_iterator: public util::iterator_facade<Derived, ValueType const, std::forward_iterator_tag>
#include <eof_iterator.hpp> The ‘eof_iterator’ class is useful for constructing forward iterators in cases where iterator extract data from some source and it’s easy to detect ‘eof’ – i.e. the situation where there’s no data. One apparent example is reading lines from a file.

Implementing such iterators using ‘iterator_facade’ directly would require to create class with three core operation, a couple of constructors. When using ‘eof_iterator’, the derived class should define only one method to get new value, plus a couple of constructors.

The basic idea is that iterator has ‘eof’ bit. Two iterators are equal only if both have their ‘eof’ bits set. The ‘get’ method either obtains the new value or sets the ‘eof’ bit.

Specifically, derived class should define:

1. A default constructor, which creates iterator with ‘eof’ bit set. The constructor body should call ‘found_eof’ method defined here.
2. Some other constructor. It should initialize some ‘data pointer’ used in iterator operation and then call ‘get’.
3. The ‘get’ method. It should operate this way:
   • look at some ‘data pointer’ to see if new element is available; if not, it should call ‘found_eof’.
   • extract new element and store it at location returned by the ‘value’ method.
   • advance the data pointer.

Essentially, the ‘get’ method has the functionality of both ‘increment’ and ‘dereference’. It’s very good for the cases where data extraction implicitly moves data pointer, like for stream operation.

Public Functions

eof_iterator ()

Protected Functions

ValueType &value ()
   Returns the reference which should be used by derived class to store the next value.

void found_eof ()
   Should be called by derived class to indicate that it can’t produce next element.

Private Functions

void increment ()

bool equal (const eof_iterator &other) const

const ValueType &dereference () const
Private Members

bool m_at_eof
ValueType m_value

Friends

friend hpx::program_options::hpx::util::iterator_core_access

namespace hpx

namespace program_options

Functions

std::string strip_prefixes (const std::string &text)

class ambiguous_option : public hpx::program_options::error_with_no_option_name
#include <errors.hpp> Class thrown when there’s ambiguity among several possible options.

Public Functions

ambiguous_option (const std::vector<std::string> &alternatives)
~ambiguous_option ()
const std::vector<std::string> &alternatives () const

Protected Functions

void substitute_placeholders (const std::string &error_template) const
Makes all substitutions using the template

Private Members

std::vector<std::string> m_alternatives

class error : public logic_error
#include <errors.hpp> Base class for all errors in the library.
Subclassed by hpx::program_options::duplicate_option_error, hpx::program_options::error_with_option_name, hpx::program_options::invalid_command_line_style, hpx::program_options::reading_file, hpx::program_options::too_many_positional_options_error
Public Functions

error (const std::string &xwhat)

class error_with_no_option_name : public hpx::program_options::error_with_option_name
#include <errors.hpp> Base class of un-parsable options, when the desired option cannot be identified.

It makes no sense to have an option name, when we can’t match an option to the parameter

Having this a part of the error_with_option_name hierarchy makes error handling a lot easier, even if
the name indicates some sort of conceptual dissonance!

Subclassed by hpx::program_options::ambiguous_option, hpx::program_options::unknown_option

Public Functions

error_with_no_option_name (const std::string &template_, const std::string &original_token = "")

void set_option_name (const std::string &)
   Does NOT set option name, because no option name makes sense

~error_with_no_option_name ()

class error_with_option_name : public hpx::program_options::error
#include <errors.hpp> Base class for most exceptions in the library.

Substitutes the values for the parameter name placeholders in the template to create the human readable error message

Placeholders are surrounded by % signs: example% Poor man’s version of boost::format

If a parameter name is absent, perform default substitutions instead so ugly placeholders are never left in-place.

Options are displayed in “canonical” form This is the most unambiguous form of the parsed option name and would correspond to option_description::format_name() i.e. what is shown by print_usage()

The “canonical” form depends on whether the option is specified in short or long form, using dashes or slashes or without a prefix (from a configuration file)

Subclassed by hpx::program_options::invalid_syntax, hpx::program_options::multiple_values, hpx::program_options::validation_error

Public Functions

error_with_option_name (const std::string &template_, const std::string &option_name = ", const std::string &original_token = "", int option_style = 0)

~error_with_option_name ()
   gcc says that throw specification on dtor is loosened without this line
void **set_substitute** (const std::string &parameter_name, const std::string &value)

Substitute parameter_name->value to create the error message from the error template

void **set_substitute_default** (const std::string &parameter_name, const std::string &value, const std::string &from, const std::string &to)

If the parameter is missing, then make the from->to substitution instead

void **add_context** (const std::string &option_name, const std::string &original_token, int option_style)

Add context to an exception

void **set_prefix** (int option_style)

**virtual** void **set_option_name** (const std::string &option_name)

Overridden in error_with_no_option_name

std::string **get_option_name** () const

**void** **set_original_token** (const std::string &original_token)

**const** char * **what** () const

Creates the error_message on the fly Currently a thin wrapper for substitute_placeholders()

**Public Members**

std::string **m_error_template**

template with placeholders

**Protected Types**

**using** **string_pair** = std::pair<std::string, std::string>

**Protected Functions**

**virtual** void **substitute_placeholders** (const std::string &error_template)

**const**

Makes all substitutions using the template

void **replace_token** (const std::string &from, const std::string &to)

**const**

std::string **get_canonical_option_name** () const

Construct option name in accordance with the appropriate prefix style: i.e. long dash or short slash etc

std::string **get_canonical_option_prefix** () const
Protected Attributes

int m_option_style
    can be 0 = no prefix (config file options) allow_long allow_dash_for_short allow_slash_for_short allow_long_disguise

std::map<std::string, std::string> m_substitutions
    substitutions from placeholders to values

std::map<std::string, string_pair> m_substitution_defaults

std::string m_message
    Used to hold the error text returned by what()

class invalid_bool_value : public hpx::program_options::validation_error
    #include <errors.hpp> Class thrown if there is an invalid bool value given

Public Functions

invalid_bool_value(const std::string &value)

class invalid_command_line_style : public hpx::program_options::error
    #include <errors.hpp> Class thrown when there are programming error related to style

Public Functions

invalid_command_line_style(const std::string &msg)

class invalid_command_line_syntax : public hpx::program_options::invalid_syntax
    #include <errors.hpp> Class thrown when there are syntax errors in given command line

Public Functions

invalid_command_line_syntax(kind_t kind, const std::string &option_name = "", const std::string &original_token = ",", int option_style = 0)

~invalid_command_line_syntax()

class invalid_config_file_syntax : public hpx::program_options::invalid_syntax

Public Functions

invalid_config_file_syntax(const std::string &invalid_line, kind_t kind)

~invalid_config_file_syntax()

std::string tokens() const
    Convenience functions for backwards compatibility

class invalid_option_value : public hpx::program_options::validation_error
    #include <errors.hpp> Class thrown if there is an invalid option value given
Public Functions

```cpp
invalid_option_value(const std::string &value)
invalid_option_value(const std::wstring &value)
```

class invalid_syntax : public hpx::program_options::error_with_option_name

```cpp
#include <errors.hpp> Class thrown when there’s syntax error either for command line or config file options. See derived children for concrete classes.
```

Subclassed by

```cpp
hpx::program_options::invalid_command_line_syntax,
hpx::program_options::invalid_config_file_syntax
```

Public Types

```cpp
enum kind_t

Values:

long_not_allowed = 30
long_adjacent_not_allowed
short_adjacent_not_allowed
empty_adjacent_parameter
missing_parameter
extra_parameter
unrecognized_line
```

Public Functions

```cpp
invalid_syntax(kind_t kind, const std::string &option_name = "", const std::string &original_token = "", int option_style = 0)
```

```cpp
~invalid_syntax()
```

```cpp
kind_t kind() const
```

```cpp
virtual std::string tokens() const
```

Convenience functions for backwards compatibility

Protected Functions

```cpp
std::string get_template(kind_t kind)
```

Used to convert kind_t to a related error text
Protected Attributes

kind_t m_kind

class multiple_occurrences : public hpx::program_options::error_with_option_name
#include <errors.hpp> Class thrown when there are several occurrences of an option, but user called a method which cannot return them all.

Public Functions

multiple_occurrences()
~multiple_occurrences()

class multiple_values : public hpx::program_options::error_with_option_name
#include <errors.hpp> Class thrown when there are several option values, but user called a method which cannot return them all.

Public Functions

multiple_values()
~multiple_values()

class reading_file : public hpx::program_options::error
#include <errors.hpp> Class thrown if config file can not be read

Public Functions

reading_file(const char *filename)

class required_option : public hpx::program_options::error_with_option_name
#include <errors.hpp> Class thrown when a required/mandatory option is missing

Public Functions

required_option(const std::string &option_name)
~required_option()

class too_many_positional_options_error : public hpx::program_options::error
#include <errors.hpp> Class thrown when there are too many positional options. This is a programming error.
Public Functions

\texttt{too\_many\_positional\_options\_error()}

\texttt{class\ unknown\_option:\ public hpx::program\_options::error\_with\_no\_option\_name
#include <errors.hpp>\ Class\ thrown\ when\ option\ name\ is\ not\ recognized.}

Public Functions

\texttt{unknown\_option(const std::string &original\_token = "")}

\texttt{\sim\unknown\_option()}

\texttt{class\ validation\_error:\ public hpx::program\_options::error\_with\_option\_name
#include <errors.hpp>\ Class\ thrown\ when\ value\ of\ option\ is\ incorrect.}

\texttt{Subclassed\ by\ hpx::program\_options::invalid\_bool\_value, hpx::program\_options::invalid\_option\_value}

Public Types

\texttt{enum\ kind\_t}

\texttt{\ Values:\}

\texttt{\ multiple\_values\_not\_allowed = 30}

\texttt{\ at\_least\_one\_value\_required}

\texttt{\ invalid\_bool\_value}

\texttt{\ invalid\_option\_value}

\texttt{\ invalid\_option}

Public Functions

\texttt{validation\_error(kind\_t\ kind, const std::string &option\_name = ",\ const std::string &original\_token = ",\ int option\_style = 0)}

\texttt{\sim validation\_error()}

\texttt{kind\_t\ kind()\ const}

Protected Functions

\texttt{std::string\ get\_template(kind\_t\ kind)}

\texttt{Used\ to\ convert\ kind\_t\ to\ a\ related\ error\ text}
Protected Attributes

`kind_t m_kind`

namespace hpx

namespace program_options

Typedefs

using option = basic_option<char>
using woption = basic_option<wchar_t>

template<class Char>
class basic_option

#include <option.hpp>
Option found in input source. Contains a key and a value. The key, in turn, can be a string (name of an option), or an integer (position in input source) – in case no name is specified. The latter is only possible for command line. The template parameter specifies the type of char used for storing the option’s value.

Public Functions

basic_option()
basic_option(const std::string &xstring_key, const std::vector<std::string> &xvalue)

Public Members

std::string string_key
String key of this option. Intentionally independent of the template parameter.

int position_key
Position key of this option. All options without an explicit name are sequentially numbered starting from 0. If an option has explicit name, ‘position_key’ is equal to -1. It is possible that both position_key and string_key is specified, in case name is implicitly added.

std::vector<std::basic_string<Char>> value
Option’s value

std::vector<std::basic_string<Char>> original_tokens
The original unchanged tokens this option was created from.

bool unregistered
True if option was not recognized. In that case, ‘string_key’ and ‘value’ are results of purely syntactic parsing of source. The original tokens can be recovered from the “original_tokens” member.

bool case_insensitive
True if string_key has to be handled case insensitive.

namespace hpx

namespace program_options
class duplicate_option_error: public hpx::program_options::error
   #include <options_description.hpp> Class thrown when duplicate option description is found.

Public Functions

duplicate_option_error(const std::string &xwhat)

class option_description
   #include <options_description.hpp> Describes one possible command line/config file option. There are two kinds of properties of an option. First describe it syntactically and are used only to validate input. Second affect interpretation of the option, for example default value for it or function that should be called when the value is finally known. Routines which perform parsing never use second kind of properties – they are side effect free.
   See options_description

Public Types

enum match_result
   Values:
   no_match
   full_match
   approximate_match

Public Functions

option_description()

option_description(const char *name, const value_semantic *s)
   Initializes the object with the passed data.
   Note: it would be nice to make the second parameter auto_ptr, to explicitly pass ownership. Unfortunately, it’s often needed to create objects of types derived from `value_semantic`:
   options_description d; d.add_options()("a", parameter<int>("n")->default_value(1)); Here, the static type returned by ‘parameter’ should be derived from `value_semantic`.
   Alas, derived->base conversion for auto_ptr does not really work, see http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2000/n1232.pdf http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#84
   So, we have to use plain old pointers. Besides, users are not expected to use the constructor directly.
   The ‘name’ parameter is interpreted by the following rules:
   • if there’s no ‘,”’ character in ‘name’, it specifies long name
   • otherwise, the part before ‘,”’ specifies long name and the part after – short name.

option_description(const char *name, const value_semantic *s, const char *description)
   Initializes the class with the passed data.

virtual ~option_description()
match_result match (const std::string &option, bool approx, bool long_ignore_case, bool short_ignore_case) const

Given ‘option’, specified in the input source, returns ‘true’ if ‘option’ specifies ‘this.

const std::string &key (const std::string &option) const

Returns the key that should identify the option, in particular in the variables_map class. The ‘option’ parameter is the option spelling from the input source. If option name contains ’*’, returns ‘option’ If long name was specified, it’s the long name, otherwise it’s a short name with pre-pended ‘-’.

std::string canonical_display_name (int canonical_option_style = 0) const

Returns the canonical name for the option description to enable the user to recognized a matching option. 1) For short options (‘-’, ‘!’), returns the short name prefixed. 2) For long options (‘’ / ‘-’) returns the first long name prefixed 3) All other cases, returns the first long name (if present) or the short name, un-prefixed.

const std::string &long_name () const

const std::pair<const std::string*, std::size_t> long_names () const

const std::string &description () const

Explanation of this option.

std::shared_ptr<const value_semantic> semantic () const

Semantic of option’s value.

std::string format_name () const

Returns the option name, formatted suitable for usage message.

std::string format_parameter () const

Returns the parameter name and properties, formatted suitably for usage message.

Private Functions

option_description &set_names (const char *name)

Private Members

std::string m_short_name

a one-character "switch” name - with its prefix, so that this is either empty or has length 2 (e.g. “-c”

std::vector<std::string> m_long_names

one or more names by which this option may be specified on a command-line or in a config file, which are not a single-letter switch. The names here are without any prefix.

std::string m_description

std::shared_ptr<const value_semantic> m_value_semantic

class options_description

#include <options_description.hpp> A set of option descriptions. This provides convenient interface for adding new option (the add_options) method, and facilities to search for options by name.

See here for option adding interface discussion.

See option_description
Public Functions

```cpp
options_description(unsigned line_length = m_default_line_length, unsigned min_description_length = m_default_line_length / 2)
```

Creates the instance.

```cpp
options_description(const std::string &caption, unsigned line_length = m_default_line_length, unsigned min_description_length = m_default_line_length / 2)
```

Creates the instance. The `caption` parameter gives the name of this `options_description` instance. Primarily useful for output. The `description_length` specifies the number of columns that should be reserved for the description text; if the option text encroaches into this, then the description will start on the next line.

```cpp
void add(std::shared_ptr<option_description> desc)
```

Adds new variable description. Throws duplicate_variable_error if either short or long name matches that of already present one.

```cpp
options_description &add(const options_description &desc)
```

Adds a group of option description. This has the same effect as adding all option_descriptions in ‘desc’ individually, except that output operator will show a separate group. Returns *this.

```cpp
std::size_t get_option_column_width() const
```

Find the maximum width of the option column, including options in groups.

```cpp
options_description_easy_init add_options()
```

Returns an object of implementation-defined type suitable for adding options to `options_description`. The returned object will have overloaded operator() with parameter type matching ‘option_description’ constructors. Calling the operator will create new `option_description` instance and add it.

```cpp
const option_description &find(const std::string &name, bool approx, bool long_ignore_case = false, bool short_ignore_case = false) const
```

```cpp
const option_description *find_nothrow(const std::string &name, bool approx, bool long_ignore_case = false, bool short_ignore_case = false) const
```

```cpp
const std::vector<std::shared_ptr<option_description>> &options() const
```

Outputs ‘desc’ to the specified stream, calling ‘f’ to output each `option_description` element.

Public Static Attributes

```cpp
const unsigned m_default_line_length
```
Private Types

using name2index_iterator = std::map<std::string, int>::const_iterator
using approximation_range = std::pair<name2index_iterator, name2index_iterator>

Private Members

std::string m_caption
const std::size_t m_line_length
const std::size_t m_min_description_length
std::vector<std::shared_ptr<option_description>> m_options
std::vector<char> belong_to_group
std::vector<std::shared_ptr<option_description>> groups

Friends

std::ostream &operator<<(std::ostream &os, const options_description &desc)

Produces a human readable output of ‘desc’, listing options, their descriptions and allowed parameters. Other options_description instances previously passed to add will be output separately.

class options_description_easy_init
#include <options_description.hpp> Class which provides convenient creation syntax to option_description.

Public Functions

options_description_easy_init(options_description *owner)

options_description_easy_init &operator()(const char *name, const char *description)

options_description_easy_init &operator()(const char *name, const value_semantic *s)

options_description_easy_init &operator()(const char *name, const value_semantic *s, const char *description)

Private Members

options_description *owner

namespace hpx

namespace program_options
## Typedefs

using parsed_options = basic_parsed_options<char>

using wparsed_options = basic_parsed_options<wchar_t>

using ext_parser = std::function<std::pair<std::string, std::string>(const std::string&)>  
Augments basic_parsed_options<wchar_t> with conversion from 'parsed_options'

using command_line_parser = basic_command_line_parser<char>
using wcommand_line_parser = basic_command_line_parser<wchar_t>

## Enums

enum collect_unrecognized_mode
Controls if the ‘collect_unregistered’ function should include positional options, or not.

Values:

- include_positional
- exclude_positional

## Functions

```cpp
template<class Char>
basic_parsed_options<Char> parse_command_line(int argc, const Char *const argv[],
    const options_description&, int style = 0, std::function<std::pair<std::string,
    std::string> (const std::string&)> ext = ext_parser())
```

Creates instance of ‘command_line_parser’, passes parameters to it, and returns the result of calling the ‘run’ method.

```cpp
template<class Char>
basic_parsed_options<Char> parse_config_file(std::basic_istream<Char>&,
    const options_description&, bool allow_unregistered = false)
```

Parse a config file.

Read from given stream.

```cpp
template<class Char = char>
basic_parsed_options<Char> parse_config_file(const char *filename,
    const options_description&, bool allow_unregistered = false)
```

Parse a config file.

Read from file with the given name. The character type is passed to the file stream.

```cpp
template<class Char>
std::vector<std::basic_string<Char>> collect_unrecognized(const
    std::vector<basic_option<Char>>& options, enum collect_unrecognized_mode
    mode)
```

Collects the original tokens for all named options with ‘unregistered’ flag set. If ‘mode’ is ‘include_positional’ also collects all positional options. Returns the vector of original tokens for all collected options.
parsed_options parse_environment (const options_description&, const std::function<std::string> std::string & name_mapper) Parse environment.

For each environment variable, the ‘name_mapper’ function is called to obtain the option name. If it returns empty string, the variable is ignored.

This is done since naming of environment variables is typically different from the naming of command line options.

parsed_options parse_environment (const options_description&, const std::string & prefix) Parse environment.

Takes all environment variables which start with ‘prefix’. The option name is obtained from variable name by removing the prefix and converting the remaining string into lower case.

parsed_options parse_environment (const options_description&, const char * prefix) This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts. This function exists to resolve ambiguity between the two above functions when second argument is of ‘char*’ type. There’s implicit conversion to both std::function and string.

std::vector<std::string> split_unix (const std::string & cmdline, const std::string & separator = " 	", const std::string & quote = "\"", const std::string & escape = \"\") Splits a given string to a collection of single strings which can be passed to command_line_parser. The second parameter is used to specify a collection of possible separator chars used for splitting. The separator is defaulted to space " ". Splitting is done in a unix style way, with respect to quotes "" and escape characters \.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class Char>
class basic_command_line_parser: private cmdline
#include <parsers.hpp> Command line parser.

The class allows one to specify all the information needed for parsing and to parse the command line. It is primarily needed to emulate named function parameters – a regular function with 5 parameters will be hard to use and creating overloads with a smaller number of parameters will be confusing.

For the most common case, the function parse_command_line is a better alternative.

There are two typedefs – command_line_parser and wcommand_line_parser, for charT == char and charT == wchar_t cases.

Public Functions

basic_command_line_parser (const std::vector<std::basic_string<Char>> & args) Creates a command line parser for the specified arguments list. The ‘args’ parameter should not include program name.

basic_command_line_parser (int argc, const Char * const argv[]) Creates a command line parser for the specified arguments list. The parameters should be the same as passed to ‘main’.
basic_command_line_parser &options(const options_description &desc)
Sets options descriptions to use.

basic_command_line_parser &positional(const positional_options_description &desc)
Sets positional options description to use.

basic_command_line_parser &style(int)
Sets the command line style.

basic_command_line_parser &extra_parser(ext_parser)
Sets the extra parsers.

basic_parsed_options<Char> run()
Parses the options and returns the result of parsing. Throws on error.

basic_command_line_parser &allow_unregistered()
Specifies that unregistered options are allowed and should be passed though. For each command
like token that looks like an option but does not contain a recognized name, an instance of ba-
sic_option<charT> will be added to result, with 'unrecognized' field set to 'true'. It’s possible to
collect all unrecognized options with the 'collect_unrecognized' function.

basic_command_line_parser &extra_style_parser(style_parser s)

Private Members

const options_description *m_desc

template<class Char>
class basic_parsed_options
#include <parsers.hpp> Results of parsing an input source. The primary use of this class is passing
information from parsers component to value storage component. This class does not makes much
sense itself.

Public Functions

basic_parsed_options(const options_description *xdescription, int options_prefix = 0)

Public Members

std::vector<basic_option<Char>> options
Options found in the source.

const options_description *description
Options description that was used for parsing. Parsers should return pointer to the instance of
option_description passed to them, and issues of lifetime are up to the caller. Can be NULL.

int m_options_prefix
Mainly used for the diagnostic messages in exceptions. The canonical option prefix for the parser
which generated these results, depending on the settings for basic_command_line_parser::style() or
cmdline::style(). In order of precedence of command_line_style enums: allow_long al-
low_long_disguise allow_dash_for_short allow_slash_for_short
class basic_parsed_options<wchar_t>
#include <parsers.hpp> Specialization of basic_parsed_options which:
• provides convenient conversion from basic_parsed_options<char>
• stores the passed char-based options for later use.

Public Functions

basic_parsed_options(const basic_parsed_options<char> &po) Constructs wrapped options from options in UTF8 encoding.

Public Members

std::vector<basic_option<wchar_t>> options
const options_description *description
basic_parsed_options<char> utf8_encoded_options Stores UTF8 encoded options that were passed to constructor, to avoid reverse conversion in some cases.

int m_options_prefix Mainly used for the diagnostic messages in exceptions. The canonical option prefix for the parser which generated these results, depending on the settings for basic_command_line_parser::style() or cmdline::style(). In order of precedence of command_line_style enums: allow_long allow_long_disguise allow_dash_for_short allow_slash_for_short

namespace hpx

namespace program_options

class positional_options_description
#include <positional_options.hpp> Describes positional options.
The class allows to guess option names for positional options, which are specified on the command line and are identified by the position. The class uses the information provided by the user to associate a name with every positional option, or tell that no name is known.
The primary assumption is that only the relative order of the positional options themselves matters, and that any interleaving ordinary options don’t affect interpretation of positional options.
The user initializes the class by specifying that first N positional options should be given the name X1, following M options should be given the name X2 and so on.

Public Functions

positional_options_description() positional_options_description &add(const char *name, int max_count) Species that up to ‘max_count’ next positional options should be given the ‘name’. The value of ‘-1’ means ‘unlimited’. No calls to ‘add’ can be made after call with ‘max_value’ equal to ‘-1’.

unsigned max_total_count() const Returns the maximum number of positional options that can be present. Can return (numeric_limits<unsigned>::max)() to indicate unlimited number.
const std::string &name_for_position(unsigned position) const

Returns the name that should be associated with positional options at 'position'. Precondition: position < max_total_count()

Private Members

std::vector<std::string> m_names
std::string m_trailing

namespace hpx

namespace program_options

Functions

template<class T>
typed_value<T> *value()

Creates a typed_value<T> instance. This function is the primary method to create value_semantic instance for a specific type, which can later be passed to 'option_description' constructor. The second overload is used when it’s additionally desired to store the value of option into program variable.

template<class T>
typed_value<T> *value(T *v)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class T>
typed_value<T, wchar_t> *wvalue()

Creates a typed_value<T> instance. This function is the primary method to create value_semantic instance for a specific type, which can later be passed to 'option_description' constructor.

template<class T>
typed_value<T, wchar_t> *wvalue(T *v)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

typed_value<bool> *bool_switch()

Works the same way as the 'value<bool>' function, but the created value_semantic won’t accept any explicit value. So, if the option is present on the command line, the value will be ‘true’.

typed_value<bool> *bool_switch(bool *v)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class T, class Char = char>
class typed_value : public hpx::program_options::value_semantic_codevt_helper<Char>, public hpx::program_options::value_semantic_codecvt_helper<Char> {
    #include <value_semantic.hpp>
}

Class which handles value of a specific type.
Public Functions

typed_value(T *store_to)
Ctor. The `store_to` parameter tells where to store the value when it’s known. The parameter can be NULL.

typed_value *default_value(const T &v)
Specifies default value, which will be used if none is explicitly specified. The type `T` should provide operator<< for ostream.

typed_value *default_value(const T &v, const std::string &textual)
Specifies default value, which will be used if none is explicitly specified. Unlike the above overload, the type `T` need not provide operator<< for ostream, but textual representation of default value must be provided by the user.

typed_value *implicit_value(const T &v)
Specifies an implicit value, which will be used if the option is given, but without an adjacent value. Using this implies that an explicit value is optional,

typed_value *value_name(const std::string &name)
Specifies the name used to to the value in help message.

typed_value *implicit_value(const T &v, const std::string &textual)
Specifies an implicit value, which will be used if the option is given, but without an adjacent value. Using this implies that an implicit value is optional, but if given, must be strictly adjacent to the option, i.e. `-o=value` or `option=value`. Giving `-o` or `option` will cause the implicit value to be applied. Unlike the above overload, the type `T` need not provide operator<< for ostream, but textual representation of default value must be provided by the user.

typed_value *notifier(std::function<void(T &)> f)
Specifies a function to be called when the final value is determined.

typed_value *composing()
Specifies that the value is composing. See the ‘is_composing’ method for explanation.

typed_value *multitoken()
Specifies that the value can span multiple tokens.

typed_value *zero_tokens()
Specifies that no tokens may be provided as the value of this option, which means that only presence of the option is significant. For such option to be useful, either the ‘validate’ function should be specialized, or the ‘implicit_value’ method should be also used. In most cases, you can use the ‘bool_switch’ function instead of using this method.

typed_value *required()
Specifies that the value must occur.

std::string name() const
Returns the name of the option. The name is only meaningful for automatic help message.

bool is_composing() const
Returns true if values from different sources should be composed. Otherwise, value from the first source is used and values from other sources are discarded.

unsigned min_tokens() const
The minimum number of tokens for this option that should be present on the command line.
unsigned `max_tokens()` const
The maximum number of tokens for this option that should be present on the command line.

bool `is_required()` const
Returns true if value must be given. Non-optional value

void `xparse(hpx::any_nonser &value_store, const std::vector<std::basic_string<Char>> &new_tokens) const`
Creates an instance of the ‘validator’ class and calls its operator() to perform the actual conversion.

virtual bool `apply_default(hpx::any_nonser &value_store) const`
If default value was specified via previous call to ‘default_value’, stores that value into ‘value_store’. Returns true if default value was stored.

void `notify(const hpx::any_nonser &value_store) const`
If an address of variable to store value was specified when creating *this, stores the value there. Otherwise, does nothing.

const `std::type_info &value_type() const`

Private Members

T *`m_store_to`
`std::string m_value_name`
`hpx::any_nonser m_default_value`
`std::string m_default_value_as_text`
`hpx::any_nonser m_implicit_value`
`std::string m_implicit_value_as_text`
`bool m_composing`
`bool m_implicit`
`bool m_multitoken`
`bool m_zero_tokens`
`bool m_required`
`std::function<void(const T&)> m_notifier`

class `typed_value_base`

#include <value_semantic.hpp>

Base class for all option that have a fixed type, and are willing to announce this type to the outside world. Any ‘value_semantics’ for which you want to find out the type can be dynamic_cast-ed to `typed_value_base`. If conversion succeeds, the ‘type’ method can be called.

Subclassed by `hpx::program_options::typed_value<T, Char>`
Public Functions

```cpp
virtual const std::type_info & value_type() const = 0
virtual ~typed_value_base()
```

class untyped_value : public hpx::program_options::value_semantic_codecvt_helper<char>

```cpp
#include <value_semantic.hpp>
```

Class which specifies a simple handling of a value: the value will have string type and only one token is allowed.

Public Functions

```cpp
untyped_value(bool zero_tokens = false)
```

```cpp
std::string name() const
```

Returns the name of the option. The name is only meaningful for automatic help message.

```cpp
unsigned min_tokens() const
```

The minimum number of tokens for this option that should be present on the command line.

```cpp
unsigned max_tokens() const
```

The maximum number of tokens for this option that should be present on the command line.

```cpp
bool is_composing() const
```

Returns true if values from different sources should be composed. Otherwise, value from the first source is used and values from other sources are discarded.

```cpp
bool is_required() const
```

Returns true if value must be given. Non-optional value

```cpp
void xparse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens)
```

If ‘value_store’ is already initialized, or new_tokens has more than one elements, throws. Otherwise, assigns the first string from ‘new_tokens’ to ‘value_store’, without any modifications.

```cpp
bool apply_default(hpx::any_nonser&) const
```

Does nothing.

```cpp
void notify(const hpx::any_nonser&) const
```

Does nothing.

Private Members

```cpp
bool m_zero_tokens
```

class value_semantic

```cpp
#include <value_semantic.hpp>
```

Class which specifies how the option’s value is to be parsed and converted into C++ types.

Subclassed by hpx::program_options::value_semantic_codecvt_helper< char >,
```
hpx::program_options::value_semantic_codecvt_helper< wchar_t >
```
Public Functions

virtual std::string name() const = 0
    Returns the name of the option. The name is only meaningful for automatic help message.

virtual unsigned min_tokens() const = 0
    The minimum number of tokens for this option that should be present on the command line.

virtual unsigned max_tokens() const = 0
    The maximum number of tokens for this option that should be present on the command line.

virtual bool is_composing() const = 0
    Returns true if values from different sources should be composed. Otherwise, value from the first
    source is used and values from other sources are discarded.

virtual bool is_required() const = 0
    Returns true if value must be given. Non-optional value

virtual void parse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens, bool utf8) const = 0
    Parses a group of tokens that specify a value of option. Stores the result in 'value_store', using
    whatever representation is desired. May be be called several times if value of the same option is
    specified more than once.

virtual bool apply_default(hpx::any_nonser &value_store) const = 0
    Called to assign default value to 'value_store'. Returns true if default value is assigned, and false
    if no default value exists.

virtual void notify(const hpx::any_nonser &value_store) const = 0
    Called when final value of an option is determined.

virtual ~value_semantic()

template<class Char>
    class value_semantic_codecvt_helper
        #include <value_semantic.hpp> Helper class which perform necessary character conversions in the
        'parse' method and forwards the data further.

    template<>
        class value_semantic_codecvt_helper<Char> : public hpx::program_options::value_semantic
            #include <value_semantic.hpp> Helper conversion class for values that accept ascii strings as in-
            put. Overrides the 'parse' method and defines new 'xparse' method taking std::string. Depending
            on whether input to parse is ascii or UTF8, will pass it to xparse unmodified, or with UTF8->ascii
            conversion.
            Subclassed by hpx::program_options::typed_value< T, Char >,
            hpx::program_options::untyped_value
Protected Functions

```cpp
virtual void xparse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens) const = 0
```

Private Functions

```cpp
void parse(hpx::any_nonser &value_store, const std::vector<std::string> &new_tokens,
           bool utf8) const
```

Parses a group of tokens that specify a value of option. Stores the result in ‘value_store’, using
whatever representation is desired. May be be called several times if value of the same option is
specified more than once.

```cpp
namespace hpx

namespace program_options

Functions

void store(const basic_parsed_options<char> &options, variables_map &m, bool utf8 = false)
Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an
option, that value is not changed, even if ‘options’ specify some value.

void store(const basic_parsed_options<wchar_t> &options, variables_map &m)
Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an
option, that value is not changed, even if ‘options’ specify some value. This is wide character variant.

void notify(variables_map &m)
Runs all ‘notify’ function for options in ‘m’.
```
class abstract_variables_map
#include <variables_map.hpp> Implements string->string mapping with convenient value casting facilities.
Subclassed by hpx::program_options::variables_map

Public Functions

abstract_variables_map()
abstract_variables_map(const abstract_variables_map *next)
virtual ~abstract_variables_map()

const variable_value &operator[](const std::string &name) const
Obtains the value of variable ‘name’, from *this and possibly from the chain of variable maps.

• if there’s no value in *this.
  – if there’s next variable map, returns value from it
  – otherwise, returns empty value
• if there’s defaulted value
  – if there’s next variable map, which has a non-defaulted value, return that
  – otherwise, return value from *this
• if there’s a non-defaulted value, returns it.

void next (abstract_variables_map *next)
Sets next variable map, which will be used to find variables not found in *this.

Private Functions

virtual const variable_value &get (const std::string &name) const = 0
Returns value of variable ‘name’ stored in *this, or empty value otherwise.

Private Members

const abstract_variables_map *m_next

class variable_value
#include <variables_map.hpp> Class holding value of option. Contains details about how the value is set and allows to conveniently obtain the value.

Public Functions

variable_value()

variable_value(const hpx::any_nonser &xv, bool xdefaulted)

template<class T>
const T &as () const
If stored value if of type T, returns that value. Otherwise, throws boost::bad_any_cast exception.
This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

bool empty() const

Returns true if no value is stored.

bool defaulted() const

Returns true if the value was not explicitly given, but has default value.

`hpx::any_nonser &value()` const

Returns the contained value.

`hpx::any_nonser &value()`

Returns the contained value.

**Private Members**

`hpx::any_nonser v`

bool m_defaulted

`std::shared_ptr<const value_semantic> m_value_semantic`

**Friends**

`friend hpx::program_options::variables_map`

void store(const basic_parsed_options<char> &options, variables_map &m, bool utf8)

Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an option, that value is not changed, even if ‘options’ specify some value.

**Public Functions**

`variables_map()`

`variables_map(const abstract_variables_map *next)`

`const variable_value &operator[](const std::string &name) const`

void clear() const

void notify()
**Private Functions**

```cpp
class variable_value &get (const std::string &name) const
```
Implementation of abstract_variables_map::get which does ‘find’ in *this.

**Private Members**

```cpp
std::set<std::string> m_final
```
Names of option with ‘final’ values – which should not be changed by subsequence assignments.

```cpp
std::map<std::string, std::string> m_required
```
Names of required options, filled by parser which has access to options_description. The map values are the “canonical” names for each corresponding option. This is useful in creating diagnostic messages when the option is absent.

**Friends**

```cpp
void store (const basic_parsed_options<char> &options, variables_map &xm, bool utf8)
```
Stores in ‘m’ all options that are defined in ‘options’. If ‘m’ already has a non-defaulted value of an option, that value is not changed, even if ‘options’ specify some value.

**Defines**

**HPX_PROGRAM_OPTIONS_VERSION**
The version of the source interface. The value will be incremented whenever a change is made which might cause compilation errors for existing code.

**HPX_PROGRAM_OPTIONS_IMPLICIT_VALUE_NEXT_TOKEN**

**properties**

The contents of this module can be included with the header hpx/modules/properties.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/properties.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
```

```cpp
namespace experimental
```

**Variables**

```cpp
hpx::experimental::prefer_t prefer
```

```cpp
struct prefer_t : public hpx::functional::detail::tag_fallback<prefer_t>
```

---

2.8. API reference 1057
Friends

template<
typename Tag,
typename ...Tn>
friend constexpr auto tag_fallback_invoke(prefer_t, Tag const &tag, Tn&&... tn)

template<
typename Tag,
typename T0,
typename ...Tn>
friend constexpr auto tag_fallback_invoke(prefer_t, Tag, T0 &&t0, Tn&&...)

resiliency

The contents of this module can be included with the header hpx/modules/resiliency.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/resiliency.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace resiliency

namespace experimental

Functions

template<
typename Pred,
typename F,
typename ...Ts>
 hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke(async_replay_validate_t, std::size_t n, Pred &&pred, F &&f, Ts&&... ts)

template<
typename F,
typename ...Ts>
 hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke(async_replay_t, std::size_t n, F &&f, Ts&&... ts)
Functions

template<
typename Executor, typename Pred, typename F, typename ... Ts>
decltype(auto) tag_invoke (async_replay_validate_t, Executor &&exec, std::size_t n, Pred &&pred, F & &f, Ts & &... ts)

namespace hpx

amespace resiliency

namespace experimental

Functions

template<
typename Vote, typename Pred, typename F, typename ... Ts>

hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke (async_replicate_vote_validate_t, std::size_t n, Vote &&vote, Pred &&pred, F &&f, Ts & &... ts)

template<
typename Vote, typename F, typename ... Ts>

hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke (async_replicate_vote_t, std::size_t n, Vote &&vote, F &&f, Ts & &... ts)

template<
typename Pred, typename F, typename ... Ts>

hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> tag_invoke (async_replicate_validate_t, std::size_t n, Pred &&pred, F &&f, Ts & &... ts)

template<
typename F, typename ... Ts>

namespace hpx

namespace resiliency

namespace experimental

Functions

template<
typename Executor, typename Vote, typename Pred, typename F, typename ...Ts>
decltype(auto) tag_invoke(
async_replicate_t,
Executor &&exec, std::size_t n, Vote &&vote, Pred &&pred, F &&f, Ts&&... ts)

template<
typename Executor, typename Vote, typename F, typename ...Ts>
decltype(auto) tag_invoke(
async_replicate_vote_validate_t, Executor &&exec, std::size_t n, Vote &&vote, F &&f, Ts&&... ts)

template<
typename Executor, typename Pred, typename F, typename ...Ts>
decltype(auto) tag_invoke(
async_replicate_validate_t, Executor &&exec, std::size_t n, Pred &&pred, F &&f, Ts&&... ts)

template<
typename Executor, typename F, typename ...Ts>
decltype(auto) tag_invoke(
async_replicate_t, Executor &&exec, std::size_t n, F &&f, Ts&&... ts)

namespace hpx

namespace resiliency

namespace experimental

Functions

template<
typename BaseExecutor, typename Validate>
replay_executor<BaseExecutor, typename std::decay<Validate>::type> make_replay_executor(BaseExecutor &&exec, std::size_t n, Validate &&validate)

template<
typename BaseExecutor>

replayExecutor<BaseExecutor, detail::replay_validator> make_replay_executor(BaseExecutor &exec, std::size_t n)

template<typename BaseExecutor, typename Validate>
class replay_executor

Public Types

template<>
using execution_category = typename BaseExecutor::execution_category
template<>
using executor_parameters_type = typename BaseExecutor::executor_parameters_type
template<typename Result>
using future_type = typename hpx::parallel::execution::executor_future<BaseExecutor, Result>::type

Public Functions

template<typename F>
replay_executor(BaseExecutor &exec, std::size_t n, F &&f)

bool operator==(replay_executor const &rhs) const

bool operator!=(replay_executor const &rhs) const

replay_executor const &context() const

template<typename F, typename ...Ts>
decltype(auto) async_execute(F &&f, Ts&&... ts) const

template<typename F, typename S, typename ...Ts>
decltype(auto) bulk_async_execute(F &&f, S const &shape, Ts&&... ts) const

Public Static Attributes

cconstexpr int num_spread = 4

cconstexpr int num_tasks = 128

Private Members

BaseExecutor &exec_

std::size_t replay_count_

Validate validator_

namespace hpx

namespace resiliency

namespace experimental
Functions

```cpp
template<typename BaseExecutor, typename Voter, typename Validate>
replicate_executor<BaseExecutor, typename std::decay<Voter>::type, typename std::decay<Validate>::type> make_replicate_executor(BaseExecutor& exec, std::size_t n, Voter&& voter, Validate&& validate);
```

```cpp
template<typename BaseExecutor, typename Validate>
replicate_executor<BaseExecutor, detail::replicate_voter, typename std::decay<Validate>::type> make_replicate_executor(BaseExecutor& exec, std::size_t n, Validate&& validate);
```

```cpp
template<typename BaseExecutor>
replicate_executor<BaseExecutor, detail::replicate_voter, detail::replicate_validator> make_replicate_executor(BaseExecutor& exec, std::size_t n);
```

Public Types

```cpp
using execution_category = typename BaseExecutor::execution_category
```

```cpp
using executor_parameters_type = typename BaseExecutor::executor_parameters_type
```

```cpp
template<typename Result>
using future_type = typename hpx::parallel::execution::executor_future<BaseExecutor, Result>::type
```
Public Functions

template<typename V, typename F>
replicate_executor (BaseExecutor &exec, std::size_t n, V &&v, F &&f)

bool operator== (replicate_executor const &rhs) const
bool operator!= (replicate_executor const &rhs) const
replicate_executor const &context () const

template<typename F, typename ...Ts>
decltype(auto) async_execute (F &&f, Ts&&... ts) const

template<typename F, typename S, typename ...Ts>
decltype(auto) bulk_async_execute (F &&f, S const &shape, Ts&&... ts) const

Public Static Attributes

cconstexpr int num_spread = 4

cconstexpr int num_tasks = 128

Private Members

BaseExecutor &exec_
std::size_t replicate_count_
Vote voter_
Validate validator_

namespace hpx

namespace resiliency

namespace experimental

Variables

hpx::resiliency::experimental::async_replay_validate_t async_replay_validate
hpx::resiliency::experimental::async_replay_t async_replay
hpx::resiliency::experimental::dataflow_replay_validate_t dataflow_replay_validate
hpx::resiliency::experimental::dataflow_replay_t dataflow_replay
hpx::resiliency::experimental::async_replicate_vote_validate_t async_replicate_vote_validate
hpx::resiliency::experimental::async_replicate_vote_t async_replicate_vote
hpx::resiliency::experimental::async_replicate_validate_t async_replicate_validate
hpx::resiliency::experimental::async_replicate_t async_replicate
hpx::resiliency::experimental::dataflow_replicate_vote_validate_t dataflow_replicate_vote_validate
struct async_replicate_t : public hpx::functional::tag<async_replicate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) repeatedly. Repeat launching on error exactly \( n \) times (except if abort_replay_exception is thrown).

struct async_replicate_validate_t : public hpx::functional::tag<async_replicate_validate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) repeatedly. Verify the result of those invocations using the given predicate \( \text{pred} \). Repeat launching on error exactly \( n \) times (except if abort_replay_exception is thrown).

struct async_replicate_t : public hpx::functional::tag<async_replicate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) exactly \( n \) times concurrently. Verify the result of those invocations by checking for exception. Return the first valid result.

struct async_replicate_validate_t : public hpx::functional::tag<async_replicate_validate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) exactly \( n \) times concurrently. Verify the result of those invocations using the given predicate \( \text{pred} \). Return the first valid result.

struct async_replicate_vote_t : public hpx::functional::tag<async_replicate_vote_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) exactly \( n \) times concurrently. Verify the result of those invocations using the given predicate \( \text{pred} \). Run all the valid results against a user provided voting function. Return the valid output.

struct async_replicate_vote_validate_t : public hpx::functional::tag<async_replicate_vote_validate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) exactly \( n \) times concurrently. Verify the result of those invocations using the given predicate \( \text{pred} \). Run all the valid results against a user provided voting function. Return the valid output.

struct dataflow_replicate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_t, async_replicate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) repeatedly. Repeat launching on error exactly \( n \) times.

Delay the invocation of \( f \) if any of the arguments to \( f \) are futures.

struct dataflow_replicate validate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_t, async_replicate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) repeatedly. Repeat launching on error exactly \( n \) times.

Delay the invocation of \( f \) if any of the arguments to \( f \) are futures.

struct dataflow_replicate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_t, async_replicate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) exactly \( n \) times concurrently. Return the first valid result.

Delay the invocation of \( f \) if any of the arguments to \( f \) are futures.

struct dataflow_replicate_validate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_t, async_replicate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function \( f \) exactly \( n \) times concurrently. Return the first valid result.
function $f$ exactly $n$ times concurrently. Verify the result of those invocations using the given predicate $pred$. Return the first valid result.

Delay the invocation of $f$ if any of the arguments to $f$ are futures.

```cpp
struct dataflow_replicate_vote_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_vote_t, async_replicate_vote_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function $f$ exactly $n$ times concurrently. Run all the valid results against a user provided voting function. Return the valid output.

Delay the invocation of $f$ if any of the arguments to $f$ are futures.
```

```cpp
struct dataflow_replicate_vote_validate_t : public hpx::resiliency::experimental::tag_deferred<dataflow_replicate_vote_validate_t, async_replicate_vote_validate_t>
#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function $f$ exactly $n$ times concurrently. Run all the valid results against a user provided voting function. Return the valid output.

Delay the invocation of $f$ if any of the arguments to $f$ are futures.
```

```cpp
template<typename Tag, typename BaseTag>
struct tag_deferred : public hpx::functional::tag<Tag>
```

### Friends

```cpp
template<typename ...Args>
auto tag_invoke (Tag, Args&&... args)
```

### Defines

```cpp
HPX_RESILIENCY_VERSION_FULL
HPX_RESILIENCY_VERSION_MAJOR
HPX_RESILIENCY_VERSION_MINOR
HPX_RESILIENCY_VERSION_SUBMINOR
HPX_RESILIENCY_VERSION_DATE
```

```cpp
namespace hpx
```

```cpp
namespace resiliency
```

```cpp
namespace experimental
```

### Functions

```cpp
unsigned int major_version ()
```

```cpp
unsigned int minor_version ()
```

```cpp
unsigned int subminor_version ()
```

```cpp
unsigned long full_version ()
```

```cpp
std::string full_version_str ()
```
resource_partitioner

The contents of this module can be included with the header `hpx/modules/resource_partitioner.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/resource_partitioner.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

    namespace resource

        class core

            Public Functions

                core (std::size_t id = invalid_core_id, numa_domain *domain = nullptr)
                std::vector<pu> &pus () const
                std::size_t id () const

            Private Functions

                std::vector<core> cores_sharing_numa_domain ()

            Private Members

                std::size_t id_
                numa_domain *domain_
                std::vector<pu> pus_

            Private Static Attributes

                constexpr const std::size_t invalid_core_id = std::size_t(-1)

            Friends

                friend hpx::resource::pu
                friend hpx::resource::numa_domain

        class numa_domain
Public Functions

numa_domain (std::size_t id = invalid_numa_domain_id)
std::vector<core> const &cores() const
std::size_t id() const

Private Members

std::size_t id_
std::vector<core> cores_

Private Static Attributes

constexpr const std::size_t invalid_numa_domain_id = std::size_t(-1)

Friends

friend hpx::resource::pu
friend hpx::resource::core

class partitioner

Public Functions

void create_thread_pool (std::string const &name, scheduling_policy sched = scheduling_policy::unspecified,
                        hpx::threads::policies::scheduler_mode = hpx::threads::policies::scheduler_mode::default_mode)
void create_thread_pool (std::string const &name, scheduler_function scheduler_creation)
void set_default_pool_name (std::string const &name)
const std::string &get_default_pool_name () const
void add_resource (hpx::resource::pu const &p, std::string const &pool_name, std::size_t num_threads = 1)
void add_resource (hpx::resource::pu const &p, std::string const &pool_name, bool exclusive, std::size_t num_threads = 1)
void add_resource (std::vector<hpx::resource::pu> const &pv, std::string const &pool_name, bool exclusive = true)
void add_resource (hpx::resource::core const &c, std::string const &pool_name, bool exclusive = true)
void add_resource (std::vector<hpx::resource::core> &cv, std::string const &pool_name, bool exclusive = true)
void add_resource (hpx::resource::numa_domain const &nd, std::string const &pool_name, bool exclusive = true)
void add_resource (std::vector<hpx::resource::numa_domain> const &ndv, std::string const &pool_name, bool exclusive = true)

std::vector<numa_domain> const &numa_domains() const

std::size_t get_number_requeste‌d_threads()

hpx::threads::topology const &get_topology() const

void configure_pools()

**Private Functions**

parti‌toner (resource::partitioner_mode rp‌mode, hpx::util::section rtcfg, hpx::threads::policies::detail::affinity_data affinity_data)

**Private Members**

detail::partitioner &partitioner_

class pu

**Public Functions**

pu (std::size_t id = invalid_pu_id, core *core = nullptr, std::size_t thread_occupancy = 0)

std::size_t id() const

**Private Functions**

std::vector<pu> pus_sharing_core()

std::vector<pu> pus_sharing_numa_domain()

**Private Members**

std::size_t id_

core *core_

std::size_t thread_occupancy_

std::size_t thread_occupancy_count_
Private Static Attributes

```cpp
constexpr const std::size_t invalid_pu_id = std::size_t(-1)
```

Friends

```cpp
friend hpx::resource::core
friend hpx::resource::numa_domain
```

namespace hpx

```cpp
namespace resource
```

Typedefs

```cpp
using scheduler_function = util::function_nonser<std::unique_ptr<hpx::threads::thread_pool_base> (hpx::threads::thread_pool_init_parameters, hpx::threads::policies::thread_queue_init_parameters)
```

Enums

```cpp
enum partitioner_mode
    This enumeration describes the modes available when creating a resource partitioner.
    Values:
    mode_default = 0
        Default mode.
    mode_allow_oversubscription = 1
        Allow processing units to be oversubscribed, i.e. multiple worker threads to share a single processing unit.
    mode_allow_dynamic_pools = 2
        Allow worker threads to be added and removed from thread pools.
```

```cpp
enum scheduling_policy
    This enumeration lists the available scheduling policies (or schedulers) when creating thread pools.
    Values:
    user_defined = -2
    unspecified = -1
    local = 0
    local_priority_fifo = 1
    local_priority_lifo = 2
    static_ = 3
    static_priority = 4
    abp_priority_fifo = 5
    abp_priority_lifo = 6
    shared_priority = 7
```
Functions

detail::partitioner &get_partitioner()
   May be used anywhere in code and returns a reference to the single, global resource partitioner.

bool is_partitioner_valid()
   Returns true if the resource partitioner has been initialized. Returns false otherwise.

runtime_configuration

The contents of this module can be included with the header hpx/modules/runtime_configuration.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/runtime_configuration.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

   namespace agas

   Enums

   enum service_mode
      Values:
      service_mode_invalid = -1
      service_mode_bootstrap = 0
      service_mode_hosted = 1

   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()

virtual hpx::program_options::options_description add_commandline_options() = 0

Return The module is expected to fill a options_description object with any additional command line options this component will handle.

Note This function will be executed by the runtime system during system startup.

Defines

HPX_REGISTER_COMPONENT_FACTORY (componentname)
This macro is used to register the given component factory with Hpx.Plugin. This macro has to be used for each of the component factories.

HPX_REGISTER_COMPONENT_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_COMPONENT_MODULE_DYNAMIC()

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()  

virtual bool get_component_info(std::vector<std::string> &fillini, std::string const &filepath, bool is_static = false) = 0
```

Return

Returns true if the parameter `fillini` has been successfully initialized with the registry data of all implemented in 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 components implemented in this module.

```
virtual void register_component_type() = 0
```

Return

Returns 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).

```namespace hpx

namespace util

Functions

bool handle_ini_file(section &ini, std::string const &loc)

bool handle_ini_file_env(section &ini, char const *env_var, char const *file_suffix = nullptr)

bool init_ini_data_base(section &ini, std::string &hpx_ini_file)

std::vector<std::shared_ptr<components::component_registry_base>> load_component_factory_static(util::section &ini, std::string name, hpx::util::get_fact_error_code &ec = throws)

void merge_component_inis(section &ini)
```
std::vector<std::shared_ptr<plugins::plugin_registry_base>> init_ini_data_default (std::string const &libs, section &ini, std::map<std::string, filesystem::path> &base_names, std::map<std::string, hpx::util::plugin::dll> &modules, std::vector<std::shared_ptr<components::component_registry_base>> &component_registries)

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 ()

virtual bool get_plugin_info (std::vector<std::string> &fillini) = 0
Return the configuration information for any plugin implemented by this module.

Return Returns true if the parameter fillini has been successfully initialized with the registry data of all implemented in 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.
virtual void init (int*, char***, util::runtime_configuration&)

namespace hpx

namespace util

Functions

bool register_lock_globally (void const*)
bool unregister_lock_globally (void const*)
void enable_global_lock_detection()
void disable_global_lock_detection()

namespace hpx

namespace util

class runtime_configuration: public section

Public Functions

runtime_configuration(char const *argv0, runtime_mode mode, 
std::vector<std::string> const &extra_static_ini_defs = 
{ })

void reconfigure (std::string const &ini_file)
void reconfigure (std::vector<std::string> const &ini_defs)

std::vector<std::shared_ptr<plugins::plugin_registry_base>> load_modules (std::vector<std::shared_ptr<component_registry_base>> &component_registries)

void load_components_static (std::vector<components::static_factory_load_data_type> &static_modules)
agas::service_mode get_agas_service_mode() const
std::uint32_t get_num_localities() const
void set_num_localities(std::uint32_t)
bool enable_networking() const
std::uint32_t get_first_used_core() const
void set_first_used_core(std::uint32_t)
std::size_t get_ipc_data_buffer_cache_size() const
std::size_t get_agas_local_cache_size (std::size_t dflt = HPX_AGAS_LOCAL_CACHE_SIZE) const
bool get_agas_caching_mode() const
bool get_agas_range_caching_mode() const
std::size_t get_agas_max_pending_refcnt_requests() const
bool load_application_configuration(char const *filename, error_code &ec = throws)
bool get_itt_notify_mode() const
bool enable_lock_detection() const
bool enable_global_lock_detection() const
bool enable_minimal_deadlock_detection() const
bool enable_spinlock_deadlock_detection() const
std::size_t get_spinlock_deadlock_detection_limit() const
std::size_t trace_depth() const
std::size_t get_os_thread_count() const
std::string get_cmd_line() const
std::ptrdiff_t get_default_stack_size() const
std::ptrdiff_t get_stack_size(threads::thread_stacksize stacksize) const
std::size_t get_thread_pool_size(char const *poolname) const
std::string get_endian_out() const
std::uint64_t get_max_inbound_message_size() const
std::uint64_t get_max_outbound_message_size() const
std::map<std::string, hpx::util::plugin::dll> &modules()

Public Members

runtime_mode mode_

Private Functions

std::ptrdiff_t init_stack_size(char const *entryname, char const *defaultvaluestr,
                              std::ptrdiff_t defaultvalue) const
std::ptrdiff_t init_small_stack_size() const
std::ptrdiff_t init_medium_stack_size() const
std::ptrdiff_t init_large_stack_size() const
std::ptrdiff_t init_huge_stack_size() const
void pre_initialize_ini()
void \texttt{post\_initialize\_ini}(std::string &hpx\_ini\_file, std::vector<std::string> const &cmdline\_ini\_defs)

void \texttt{pre\_initialize\_logging\_ini}()

void \texttt{reconfigure}()

void \texttt{load\_component\_paths}(std::vector<std::shared_ptr<plugins::plugin\_registry\_base>> &plugin\_registries, std::vector<std::shared_ptr<components::component\_registry\_base>> &component\_registries, std::string const &component\_base\_paths, std::string const &component\_path\_suffixes, std::set<std::string> &component\_paths, std::map<std::string, filesystem::path> &basenames)

void \texttt{load\_component\_path}(std::vector<std::shared_ptr<plugins::plugin\_registry\_base>> &plugin\_registries, std::vector<std::shared_ptr<components::component\_registry\_base>> &component\_registries, std::string const &path, std::set<std::string> &component\_paths, std::map<std::string, filesystem::path> &basenames)

private

std::string hpx\_ini\_file
std::vector<std::string> cmdline\_ini\_defs
std::vector<std::string> extra\_static\_ini\_defs
std::uint32_t num\_localities
std::uint32_t num\_os\_threads
std::ptrdiff_t small\_stacksize
std::ptrdiff_t medium\_stacksize
std::ptrdiff_t large\_stacksize
std::ptrdiff_t huge\_stacksize
bool need\_to\_call\_pre\_initialize
std::map<std::string, hpx::util::plugin::dll> modules

namespace hpx

**Enums**

define \texttt{runtime\_mode} as an enumerated type with two values: console mode and worker mode.

**Values:**

- \texttt{invalid} = -1
- \texttt{console} = 0
  - The runtime is the console locality.
- \texttt{worker} = 1
  - The runtime is a worker locality.
connect = 2
The runtime is a worker locality connecting late

local = 3
The runtime is fully local.

default_ = 4
The runtime mode will be determined based on the command line arguments

last

Functions

char const *get_runtime_mode_name (runtime_mode state)
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

Defines

HPX_DECLARE_FACTORY_STATIC (name, base)

HPX_DEFINE_FACTORY_STATIC (module, name, base)

HPX_INIT_REGISTRY_MODULE_STATIC (name, base)

HPX_INIT_REGISTRY_FACTORY_STATIC (name, componentname, base)

HPX_INIT_REGISTRY_COMMANDLINE_STATIC (name, base)

HPX_INIT_REGISTRY_STARTUP_SHUTDOWN_STATIC (name, base)

namespace hpx

namespace components

Functions

bool &get_initial_static_loading ()

std::vector<static_factory_load_data_type> &get_static_module_data ()

void init_registry_module (static_factory_load_data_type const &)

bool get_static_factory (std::string const &instance, util::plugin::get_plugins_list_type &f)

void init_registry_factory (static_factory_load_data_type const &)

2.8. API reference
bool get_static_commandline (std::string const &instance, util::plugin::get_plugins_list_type &f)

void init_registry_commandline (static_factory_load_data_type const&)

bool get_static_startup_shutdown (std::string const &instance, util::plugin::get_plugins_list_type &f)

void init_registry_startup_shutdown (static_factory_load_data_type const&)

struct static_factory_load_data_type

Public Members

char const *name

hpx::util::plugin::get_plugins_list_type get_factory

runtime_local

The contents of this module can be included with the header hpx/modules/runtime_local.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/runtime_local.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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() const

virtual bool get_startup_function/startup_function_type &startup, bool &pre_startup) = 0

Return any startup function for this component.

Return
Returns true if the parameter startup has been successfully initialized with the startup function.

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.

namespace hpx

Functions

std::string get_config_entry(const std::string &key, const std::string &dflt) const

Retrieve the string value of a configuration entry given by key.

std::string get_config_entry(const std::string &key, const std::size_t &dflt) const

Retrieve the integer value of a configuration entry given by key.

void set_config_entry(const std::string &key, const std::string &value)

Set the string value of a configuration entry given by key.

void set_config_entry(const std::string &key, const std::size_t &value)

Set the integer value of a configuration entry given by key.

void set_config_entry_callback(const std::string &key, util::function_nonser<void> const &callback)

Set the string value of a configuration entry given by key.

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.

Return  The formatted string holding all of the available diagnostic information stored in the given exception instance.

See `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`.

Exceptions

- `std::bad_alloc`: (if any of the required allocation operations fail)

`std::uint32_t get_error_locality_id(hpx::exception_info const &xi)`

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.

Return  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`.

See `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`.

Exceptions

- `nothing`:

`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.

**Return** 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.

**See**
- `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_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`.

**Exceptions**
- `std::bad_alloc`: (if one of the required allocations fails)

```cpp
std::int64_t get_error_process_id(hpx::exception_info const &xi)
```

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.

**Return** 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.

**See**
- `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_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`.

**Exceptions**
- `nothing`:

```cpp
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.

**Return** 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.

**See**
- `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`

2.8. API reference
Parameters

- \texttt{xi}: The parameter \texttt{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: \texttt{hpx::exception\_info, hpx::error\_code, std::exception, or std::exception\_ptr}.

Exceptions

- \texttt{std::bad\_alloc}: (if one of the required allocations fails)

\begin{Verbatim}
\texttt{std::string get\_error\_backtrace (hpx::exception\_info const &xi)}
\end{Verbatim}

Return the stack backtrace from the point the exception was thrown.

The function \texttt{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.

Return The stack backtrace from the point the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

See \texttt{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

- \texttt{xi}: The parameter \texttt{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: \texttt{hpx::exception\_info, hpx::error\_code, std::exception, or std::exception\_ptr}.

Exceptions

- \texttt{std::bad\_alloc}: (if one of the required allocations fails)

\begin{Verbatim}
\texttt{std::size\_t get\_error\_os\_thread (hpx::exception\_info const &xi)}
\end{Verbatim}

Return the sequence number of the OS-thread used to execute HPX-threads from which the exception was thrown.

The function \texttt{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.

Return 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).

See \texttt{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
The parameter `xi` 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`.

**Exceptions**

- **nothing:**

```cpp
std::size_t get_error_thread_id (hpx::exception_info const &xi)
```

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.

**Return** 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)`.

**See** `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_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`.

**Exceptions**

- **nothing:**

```cpp
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.

**Return** 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.

**See** `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_backtrace()`, `hpx::get_error_env()`, `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`.

**Exceptions**

- **std::bad_alloc**: (if one of the required allocations fails)
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.

Return 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.

See hpx::diagnostic_information(), hpx::get_error_host_name(), hpx::get_error_process_id(), hpx::get_error_os_thread(), hpx::get_error_file_name(), hpx::get_error_function_name(), hpx::get_error_function_name(), 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.

Exceptions
• std::bad_alloc: (if one of the required allocations fails)

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.

Return 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.

See hpx::diagnostic_information(), hpx::get_error_host_name(), hpx::get_error_process_id(), hpx::get_error_os_thread(), hpx::get_error_file_name(), hpx::get_error_function_name(), 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.

Exceptions
• std::bad_alloc: (if one of the required allocations fails)
Functions

void may_attach_debugger (std::string const & category)
Attaches a debugger if category is equal to the configuration entry hpx.attach-debugger.

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.

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.

Return 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.

See future<std::string> get_locality_name(naming::id_type const& id)

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 con-
sole at application startup.

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.
See hpx:\find\_all\_localities, hpx:\get\_num\_localities

//cos: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.

Note This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

See hpx:\find\_all\_localities, hpx:\get\_num\_localities

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.

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.

See hpx:\find\_all\_localities, hpx:\get\_num\_localities

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.

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 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>”.

namespace hpx

namespace util

class interval_timer

Public Functions

`HPX_NON_COPYABLE (interval_timer)`

`interval_timer()`

`interval_timer (util::function_nonser<bool>)`

`interval_timer (util::function_nonser<void> const &on_termstd::int64_t microsecs, std::string const &description = "", bool pre_shutdown = false)`

`interval_timer (util::function_nonser<void>)`

`interval_timer (const &hpx::chrono::steady_duration const &rel_time, char const *description = "", bool pre_shutdown = false)`

`~interval_timer()`

`bool start (bool evaluate = true)`

`bool stop (bool terminate = false)`

`bool restart (bool evaluate = true)`

`bool is_started () const`

`bool is_terminated () const`

`std::int64_t get_interval () const`

`void change_interval (std::int64_t new_interval)`

`void change_interval (hpx::chrono::steady_duration const &new_interval)`
Private Members

std::shared_ptr<detail::interval_timer> timer_

namespace hpx

namespace runtime_local

 Enums

enum os_thread_type
Types of kernel threads registered with the runtime.
Values:
unknown = -1
main_thread = 0
    kernel thread represents main thread
worker_thread
    kernel thread is used to schedule HPX threads
io_thread
    kernel thread can be used for IO operations
timer_thread
    kernel is used by timer operations
parcel_thread
    kernel is used by networking operations
custom_thread
    kernel is registered by the application

 Functions

std::string get_os_thread_type_name(os_thread_type type)
    Return a human-readable name representing one of the kernel thread types.

 struct os_thread_data
    #include <os_thread_type.hpp> Registration data for kernel threads that is maintained by the runtime internally

 Public Members

std::string label_
    name used for thread registration
std::thread::id id_
    thread id of corresponding kernel thread
std::uint64_t native_handle_
    the threads native handle
os_thread_type type_
    HPX thread type.
namespace hpx

namespace util

class pool_timer

Public Functions

HPX_NON_COPYABLE(pool_timer)

pool_timer()

pool_timer(util::function_nonser<bool>)
  > const &util::function_nonser<void> const &on_termstd::string const &description = ",
  bool pre_shutdown = true

~pool_timer()

bool start(hpx::chrono::steady_duration const &time_duration, bool evaluate = false)

bool stop()

bool is_started() const

bool is_terminated() const

Private Members

std::shared_ptr<detail::pool_timer> timer_

namespace hpx

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.

namespace hpx

namespace threads
Functions

template<typename F, typename ...Ts>
util::invoke_result<F, Ts...>::type run_as_hpx_thread(F const &f, Ts&&... vs)

namespace hpx

namespace threads

Functions

template<typename F, typename ...Ts>
hpx::future<typename util::invoke_result<F, Ts...>::type> run_as_os_thread(F &&f, Ts&&... vs)

namespace hpx

Functions

void set_error_handlers()

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)

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(util::function_nonser<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()
Return a reference to the internal PAPI thread manager.

threads::topology const &get_topology() const

virtual int run (util::function_nonser<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.

Return This function will return the value as returned as the result of the invocation of the function object given by the parameter func.

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.

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.

Return This function will always return 0 (zero).

virtual void rethrow_exception()
Rethrow any stored exception (to be called after stop())

virtual int start (util::function_nonser<hpx_main_function_type> const &func, bool blocking = false)
Start the runtime system.

Return 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.

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.
virtual int start (bool blocking = false)
Start the runtime system.

**Return** 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.

**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.

virtual int wait ()
Wait for the shutdown action to be executed.

**Return** 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 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.

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.
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 setup 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 setup 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.

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:

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.

Note This function should be called for each thread exactly once. It will fail if it is called more than once.

Return This function will return whether the requested operation succeeded or not.

```cpp
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).

Return This function will return whether the requested operation succeeded or not.

```cpp
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.

```cpp
virtual bool enumerate_os_threads (util::function_nonser<bool> runtime_local::os_thread_data const&) const
```

Enumerate all OS threads that have registered with the runtime.

```cpp
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&&) const
notification_policy_type::on_startstop_type on_stop_func (notification_policy_type::on_startstop_type&&) const
notification_policy_type::on_error_type on_error_func (notification_policy_type::on_error_type&&) const
```

```cpp
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 lcos::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 ()
```
Public Static Functions

```cpp
static std::uint64_t get_system_uptime()
```

Return the system uptime measure on the thread executing this call.

Protected Types

```cpp
using on_exit_type = std::vector<util::function_nonser<void>>
```

Protected Functions

```cpp
runtime(hpx::util::runtime_configuration &rtcfg)

void set_notification_policies(notification_policy_type &&notifier,
    threads::detail::network_background_callback_type
    network_background_callback)
```

```cpp
void init()
    Common initialization for different constructors.
```

```cpp
void init_global_data()
```

```cpp
void deinit_global_data()
```

```cpp
threads::thread_result_type run_helper(util::function_nonser<hpx::main_function_type>
    const &func, int &result, bool call_startup_functions)
```

```cpp
void wait_helper(std::mutex &mtx, std::condition_variable &cond, bool &running)
```

Protected Attributes

```cpp
on_exit_type on_exit_functions_
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_
util::io_service_pool main_pool_

notification_policy_type notifier_
```
\texttt{std::unique_ptr<hpx::threads::threadmanager> thread_manager_}

\section*{Protected Static Attributes}

\texttt{std::atomic<int> instance_number_counter_}

\section*{Private Functions}

\texttt{void stop_helper (bool blocking, std::condition_variable &cond, std::mutex &mtx)}

Helper function to stop the runtime.

\textbf{Parameters}

- \texttt{blocking}: [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is \texttt{false} then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to \texttt{true} to wait for all internal work to be completed.

\texttt{void deinit_tss_helper (char const *context, std::size_t num)}

\texttt{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)}

\texttt{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)}

\texttt{void notify_finalize ()}

\texttt{void wait_finalize ()}

\texttt{void call_startup_functions (bool pre_start)}

\section*{Private Members}

\texttt{std::list<startup_function_type> pre_startup_functions_}

\texttt{std::list<startup_function_type> startup_functions_}

\texttt{std::list<shutdown_function_type> pre_shutdown_functions_}

\texttt{std::list<shutdown_function_type> shutdown_functions_}

\texttt{bool stop_called_}

\texttt{bool stop_done_}

\texttt{std::condition_variable wait_condition_}

\texttt{namespace threads}
Functions

cchar 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)

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 (util::function_nonser<bool> os_thread_data const & > const &/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 (util::function_nonser<void>)
   > const & Register a function to be called during system shutdown.

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 hpx
namespace parallel

namespace execution

Enums

enum service_executor_type
{
  io_thread_pool
  Selects creating a service executor using the I/O pool of threads
  parcel_thread_pool
  Selects creating a service executor using the parcel pool of threads
  timer_thread_pool
  Selects creating a service executor using the timer pool of threads
  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

parcel_pool_executor(char const *name_suffix = "-tcp")

struct service_executor : public service_executor

Public Functions

service_executor(service_executor_type t, char const *name_suffix = "")

struct timer_pool_executor : public service_executor

Public Functions
Public Functions

    timer_pool_executor()

namespace hpx

Typedefs

typedef util::unique_function_nonser<void()> shutdown_function_type

The type of a 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 \texttt{hpx::finalize()} but guaranteed before any shutdown function is executed (system-wide)

Any of the functions registered with \texttt{register_pre_shutdown_function} are guaranteed to be executed by an HPX thread during the execution of \texttt{hpx::finalize()} before any of the registered shutdown functions are executed (see: \texttt{hpx::register_shutdown_function()}).

\textbf{Note} If this function is called while the pre-shutdown functions are being executed, or after that point, it will raise a invalid_status exception.

\textbf{See} \texttt{hpx::register_shutdown_function()}

Parameters

- \texttt{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 \texttt{hpx::finalize()} but guaranteed after any pre-shutdown function is executed (system-wide)

Any of the functions registered with \texttt{register_shutdown_function} are guaranteed to be executed by an HPX thread during the execution of \texttt{hpx::finalize()} after any of the registered pre-shutdown functions are executed (see: \texttt{hpx::register_pre_shutdown_function()}).

\textbf{Note} If this function is called while the shutdown functions are being executed, or after that point, it will raise a invalid_status exception.

\textbf{See} \texttt{hpx::register_pre_shutdown_function()}

Parameters

- \texttt{f}: [in] The function to be registered to run by an HPX thread as a shutdown function.
Typedefs

typedef util::unique_function_nonser<void ()> startup_function_type

The type of a function which is registered to be executed as a startup or pre-startup function.

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).

Note If this function is called while the pre-startup functions are being executed or after that point, it will raise a invalid_status exception.

Parameters

• f: [in] The function to be registered to run by an HPX thread as a pre-startup function.

See hpx::register_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).

Note If this function is called while the startup functions are being executed or after that point, it will raise a invalid_status exception.

Parameters

• f: [in] The function to be registered to run by an HPX thread as a startup function.

See hpx::register_pre_startup_function()
namespace hpx

Functions

bool threadmanager_is (state st)

bool threadmanager_is_at_least (state st)

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).

Return The currently installed error handler function.

Note This function can be called before the HPX runtime is initialized.

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).

Return The currently installed error handler function.

Note This function can be called before the HPX runtime is initialized.

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).

Return The currently installed error handler function.

Note This function can be called before the HPX runtime is initialized.

threads::policies::callback_notifier::on_startstop_type register_thread_on_start_func (threads::policies::callback_notifier::on_startstop_type &&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).

Return The previously registered function of this category. It is the user’s responsibility to call that function if the callback is invoked by HPX.

Note This function can be called before the HPX runtime is initialized.

Parameters
• \( f \): The function to install as the new start handler.

```cpp
threads::policies::callback_notifier::on_startstop_type register_thread_on_stop_func (threads::policies::callback_notifier::on_startstop_type &&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 \( \text{get_thread_on_stop_func} \)).

**Return** The previously registered function of this category. It is the user’s responsibility to call that function if the callback is invoked by HPX.

**Note** This function can be called before the HPX runtime is initialized.

**Parameters**

• \( f \): The function to install as the new stop handler.

```cpp
threads::policies::callback_notifier::on_error_type register_thread_on_error_func (threads::policies::callback_notifier::on_error_type &&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 \( \text{get_thread_on_error_func} \)).

**Return** The previously registered function of this category. It is the user’s responsibility to call that function if the callback is invoked by HPX.

**Note** This function can be called before the HPX runtime is initialized.

**Parameters**

• \( f \): The function to install as the new error handler.

```cpp
namespace hpx
```

```cpp
namespace util
```

```cpp
class thread_mapper
```

**Public Types**

```cpp
using callback_type = detail::thread_mapper_callback_type
```
**Public Functions**

```cpp
HPX_NON_COPYABLE(thread_mapper)

thread_mapper()
~thread_mapper()

std::uint32_t register_thread(char const *label, runtime_local::os_thread_type type)

bool unregister_thread()

std::uint32_t get_thread_index(std::string const &label) const

std::uint32_t get_thread_count() const

bool register_callback(std::uint32_t tix, callback_type const &)

bool revoke_callback(std::uint32_t tix)

std::thread::id get_thread_id(std::uint32_t tix) const

std::uint64_t get_thread_native_handle(std::uint32_t tix) const

std::string const &get_thread_label(std::uint32_t tix) const

runtime_local::os_thread_type get_thread_type(std::uint32_t tix) const

bool enumerate_os_threads(util::function_nonser<bool> os_thread_data const & > const & f const

os_thread_data get_os_thread_data(std::string const &label) const
```

**Public Static Attributes**

```cpp
constexpr std::uint32_t invalid_index = std::uint32_t(-1)
constexpr std::uint64_t invalid_tid = std::uint64_t(-1)
```

**Private Types**

```cpp
using mutex_type = hpx::lcos::local::spinlock
using thread_map_type = std::vector<detail::os_thread_data>
using label_map_type = std::map<std::string, std::size_t>
```

**Private Members**

```cpp
mutex_type mtx_
thread_map_type thread_map_
label_map_type label_map_
```

```cpp
namespace hpx
```

```cpp
namespace resource
```
Functions

\texttt{std::size\_t get\_num\_thread\_pools()} 
Return the number of thread pools currently managed by the \texttt{resource}\_\texttt{partitioner}

\texttt{std::size\_t get\_num\_threads()} 
Return the number of threads in all thread pools currently managed by the \texttt{resource}\_\texttt{partitioner}

\texttt{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 \texttt{resource}\_\texttt{partitioner}

\texttt{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 \texttt{resource}\_\texttt{partitioner}

\texttt{std::size\_t get\_pool\_index(std::string const \& pool\_name)} 
Return the internal index of the pool given its name.

\texttt{std::string const \& get\_pool\_name(std::size\_t pool\_index)} 
Return the name of the pool given its internal index.

\texttt{threads::thread\_pool\_base \& get\_thread\_pool(std::string const \& pool\_name)} 
Return the name of the pool given its name.

\texttt{threads::thread\_pool\_base \& get\_thread\_pool(std::size\_t pool\_index)} 
Return the thread pool given its internal index.

\texttt{bool pool\_exists(std::string const \& pool\_name)} 
Return true if the pool with the given name exists.

\texttt{bool pool\_exists(std::size\_t pool\_index)} 
Return true if the pool with the given index exists.

namespace threads

Functions

\texttt{std::int64\_t get\_thread\_count(thread\_schedule\_state state)} 
The function \texttt{get\_thread\_count} returns the number of currently known threads.

\texttt{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).

\texttt{Parameters}
- \texttt{state: [in]} This specifies the thread-state for which the number of threads should be retrieved.

\texttt{std::int64\_t get\_thread\_count(thread\_priority priority, thread\_schedule\_state state)} 
The function \texttt{get\_thread\_count} returns the number of currently known threads.

\texttt{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).

\texttt{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.

\texttt{std::int64_t get_idle_core_count()}

The function \texttt{get_idle_core_count} returns the number of currently idling threads (cores).

\texttt{mask\_type get_idle_core_mask()}

The function \texttt{get_idle_core_mask} returns a bit-mask representing the currently idling threads (cores).

\texttt{bool enumerate_threads (util::function\_nonser<bool) thread\_id\_type}
\texttt{> const &f, thread\_schedule\_state state = thread\_schedule\_state::unknown}

The function \texttt{enumerate_threads} will invoke the given function \texttt{f} for each thread with a matching thread state.

\textbf{Parameters}

• \texttt{f}: [in] The function which should be called for each matching thread. Returning ‘false’ from this function will stop the enumeration process.
• \texttt{state}: [in] This specifies the thread-state for which the threads should be enumerated.

\begin{verbatim}
namespace hpx

namespace util

namespace debug

Functions

\texttt{std::vector<hpx::threads::thread\_id\_type> get\_task\_ids (hpx::threads::thread\_schedule\_state state = hpx::threads::thread\_schedule\_state::suspended)}

\texttt{std::vector<hpx::threads::thread\_data*> get\_task\_data (hpx::threads::thread\_schedule\_state state = hpx::threads::thread\_schedule\_state::suspended)}

\texttt{std::string suspended\_task\_backtraces ()}
\end{verbatim}

\textbf{schedulers}

The contents of this module can be included with the header \texttt{hpx/modules/schedulers.hpp}. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header \texttt{hpx/modules/schedulers.hpp}, not the particular header in which the functionality you would like to use is defined. See \textit{Public API} for a list of names that are part of the public \textit{HPX} API.

\begin{verbatim}
namespace hpx

namespace threads

namespace policies
\end{verbatim}
Typedefs

using default_local_priority_queue_scheduler_terminated_queue = lockfree_fifo

template<typename Mutex = std::mutex, typename PendingQueuing = lockfree_fifo, typename StagedQueuing = lockfree_fifo, typename TerminatedQueuing = default_local_priority_queue_scheduler_terminated_queue>
class local_priority_queue_scheduler : public scheduler_base

#include <local_priority_queue_scheduler.hpp> The local_priority_queue_scheduler maintains exactly one queue of work items (threads) per OS thread, where this OS thread pulls its next work from. Additionally it maintains separate queues: several for high priority threads and one for low priority threads. High priority threads are executed by the first N OS threads before any other work is executed. Low priority threads are executed by the last OS thread whenever no other work is available.

Public Types

typedef std::false_type has_periodic_maintenance
typedef thread_queue<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing> thread_queue_type
typedef init_parameter init_parameter_type

Public Functions

local_priority_queue_scheduler(init_parameter_type const &init, bool deferred_initialization = true)

~local_priority_queue_scheduler()

void abort_all_suspended_threads()

bool cleanup_terminated (bool delete_all)

bool cleanup_terminated (std::size_t num_thread, bool delete_all)

void create_thread (thread_init_data &data, thread_id_ref_type *id, error_code &ec)

bool get_next_thread (std::size_t num_thread, bool running,
threads::thread_id_ref_type &thrd, bool enable_stealing)  
Return the next thread to be executed, return false if none is available

void schedule_thread (threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback = false, thread_priority priority = thread_priority::normal)  
Schedule the passed thread.

void schedule_thread_last (threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback = false, thread_priority priority = thread_priority::normal)

void destroy_thread (threads::thread_data *thrd)  
Destroy the passed thread as it has been terminated.

std::int64_t get_queue_length (std::size_t num_thread = std::size_t(-1)) const
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 = std::size_t(-1), bool = false) const

bool is_core_idle (std::size_t num_thread) const

bool enumerate_threads (util::function_nonser<bool>) thread_id_type > const &f, thread_schedule_state state = thread_schedule_state::unknown const

bool wait_or_add_new (std::size_t num_thread, bool running, std::int64_t &idle_loop_count, bool enable_stealing, std::size_t &added)

This is a function which gets called periodically by the thread manager to allow for maintenance tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to be terminated (i.e. no more work has to be done).

void on_start_thread (std::size_t num_thread)

void on_stop_thread (std::size_t num_thread)

void on_error (std::size_t num_thread, std::exception_ptr const &e)

void reset_thread_distribution ()

Public Static Functions

static std::string get_scheduler_name ()

Protected Attributes

std::atomic<std::size_t> curr_queue_

detail::affinity_data const &affinity_data_

const std::size_t num_queues_

const std::size_t num_high_priority_queues_

thread_queue_type low_priority_queue_

std::vector<util::cache_line_data<thread_queue_type>*>> queues_

std::vector<util::cache_line_data<thread_queue_type>*>> high_priority_queues_

std::vector<util::cache_line_data<std::vector<std::size_t>>> victim_threads_

struct init_parameter

Public Functions

template<>
init_parameter (std::size_t num_queues, detail::affinity_data const &affinity_data, std::size_t num_high_priority_queues = std::size_t(-1), thread_queue_init_parameters thread_queue_init = {}, char const *description = "local_priority_queue_scheduler")

template>
init_parameter (std::size_t num_queues, detail::affinity_data const &affinity_data, char const *description)
Public Members

```cpp
template<>
std::size_t num_queues_
```

```cpp
template<>
std::size_t num_high_priority_queues_
```

```cpp
template<>
thread_queue_init_parameters thread_queue_init_
```

```cpp
template<>
detail::affinity_data const &affinity_data_
```

```cpp
template<>
char const *description_
```

```cpp
namespace hpx
```

```cpp
namespace threads
```

```cpp
namespace policies
```

Typedefs

```cpp
using default_local_queue_scheduler_terminated_queue = lockfree_fifo
```

```cpp
template<
Mutex = std::mutex, typename PendingQueuing = lockfree_fifo, typename StagedQueuing = lockfree_fifo, typename TerminatedQueuing = default_local_queue_scheduler_terminated_queue>

class local_queue_scheduler : public scheduler_base
```

```cpp
#include <local_queue_scheduler.hpp>
```

The local_queue_scheduler maintains exactly one queue of work items (threads) per OS thread, where this OS thread pulls its next work from.

Public Types

```cpp
typedef std::false_type has_periodic_maintenance
```

```cpp
typedef thread_queue<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing> thread_queue_type
```

```cpp
typedef init_parameter init_parameter_type
```

Public Functions

```cpp
local_queue_scheduler (init_parameter_type const &init, bool deferred_initialization = true)
```

```cpp
virtual ~local_queue_scheduler ()
```

```cpp
void abort_all_suspended_threads ()
```

```cpp
bool cleanup_terminated (bool delete_all)
```

```cpp
bool cleanup_terminated (std::size_t num_thread, bool delete_all)
```

```cpp
void create_thread (thread_init_data &data, thread_id_ref_type *id, error_code &ec)
```
virtual bool get_next_thread (std::size_t num_thread, bool running, threads::thread_id_ref_type &thrd, bool)
Return the next thread to be executed, return false if none is available

void schedule_thread (threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback, thread_priority = thread_priority::normal)
Schedule the passed thread.

void schedule_thread_last (threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback, thread_priority = thread_priority::normal)

void destroy_thread (threads::thread_data *thrd)
Destroy the passed thread as it has been terminated.

std::int64_t get_queue_length (std::size_t num_thread = std::size_t(-1)) const

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 = std::size_t(-1), bool = false) const

bool is_core_idle (std::size_t num_thread) const

bool enumerate_threads (util::function_nonser<bool> thread_id_type > const &kf, thread_schedule_state state = thread_schedule_state::unknown const

virtual bool wait_or_add_new (std::size_t num_thread, bool running, std::int64_t &idle_loop_count, bool, std::size_t &added)
This is a function which gets called periodically by the thread manager to allow for maintenance tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to be terminated (i.e. no more work has to be done).

void on_start_thread (std::size_t num_thread)

void on_stop_thread (std::size_t num_thread)

void on_error (std::size_t num_thread, std::exception_ptr const &e)

Public Static Functions

static std::string get_scheduler_name ()

Protected Attributes

std::vector<thread_queue_type*> queues_
std::atomic<std::size_t> curr_queue_
detail::affinity_data const &affinity_data_
mask_type steals_in_numa_domain_
mask_type steals_outside_numa_domain_
std::vector<mask_type> numa_domain_masks_
std::vector<mask_type> outside_numa_domain_masks_
struct init_parameter

Public Functions

template<>
init_parameter (std::size_t num_queues, detail::affinity_data const &affinity_data,
               thread_queue_init_parameters thread_queue_init = {}, char const *description = "local_queue_scheduler")

template<>
init_parameter (std::size_t num_queues, detail::affinity_data const &affinity_data,
               char const *description)

Public Members

template<>
std::size_t num_queues_

template<>
thread_queue_init_parameters thread_queue_init_

template<>
detail::affinity_data const &affinity_data_

template<>
char const *description_

namespace hpx

namespace threads

namespace policies

struct concurrentqueue_fifo

template<typename T>
struct apply

Public Types

template<>
using type = moodycamel_fifo_backend<T>

struct lockfree_fifo

template<typename T>
struct apply
Public Types

\[
\text{template<typename T>}
\text{\textbf{using\ type} = lockfree_fifo_backend<T>}
\]

\[
\text{template<typename T>}
\text{\textbf{struct\ lockfree_fifo_backend}}
\]

Public Types

\[
\text{template<typename T>}
\text{\textbf{using\ container\ type} = boost::lockfree::queue<T, hpx::util::aligned_allocator<T> >}
\]

\[
\text{template<typename T>}
\text{\textbf{using\ value\ type} = T}
\]

\[
\text{template<typename T>}
\text{\textbf{using\ reference} = T&}
\]

\[
\text{template<typename T>}
\text{\textbf{using const\ reference} = T const&}
\]

\[
\text{template<typename T>}
\text{\textbf{using rvalue\ reference} = T&&}
\]

\[
\text{template<typename T>}
\text{\textbf{using size\ type} = std::uint64_t}
\]

Public Functions

\[
\text{lockfree_fifo_backend (size\ type initial\ size = 0, size\ type = size\ type(-1))}
\]

bool push (const\ reference\ val, bool = false)

bool push (rvalue\ reference\ val, bool = false)

bool pop (reference\ val, bool = true)

bool empty ()

Private Members

container\ type queue_

\[
\text{template<typename T>}
\text{\textbf{struct\ moodycamel_fifo_backend}}
\]
Public Types

template<>
using container_type = hpx::concurrency::ConcurrentQueue<T>

template<>
using value_type = T

template<>
using reference = T&

template<>
using const_reference = T const&

template<>
using rvalue_reference = T&&

template<>
using size_type = std::uint64_t

Public Functions

moodycamel_fifo_backend(size_type initial_size = 0, size_type = size_type(-1))

bool push(const_reference val, bool = false)

bool push(rvalue_reference val, bool = false)

bool pop(reference val, bool = true)

bool empty()

Private Members

class container_type queue_

Defines

QUEUE_HOLDER_NUMA_DEBUG

namespace hpx

Functions

static hpx::debug::enable_print<QUEUE_HOLDER_NUMA_DEBUG> hpx::nq_deb("QH_NUMA")

namespace threads

namespace policies

template<
    typename QueueType>
struct queue_holder_numa
Public Types

```cpp
template<>
using ThreadQueue = queue_holder_thread<QueueType>

template<>
using mutex_type = typename QueueType::mutex_type
```

Public Functions

```cpp
queue_holder_numa ()
~queue_holder_numa ()

void init (std::size_t domain, std::size_t queues)

std::size_t size () const

ThreadPool *thread_queue (std::size_t id) const

bool get_next_thread_HP (std::size_t qidx, threads::thread_id_ref_type &thrd, bool stealing, bool core_stealing)

bool get_next_thread (std::size_t qidx, threads::thread_id_ref_type &thrd, bool stealing, bool core_stealing)

bool add_new_HP (ThreadPool *receiver, std::size_t qidx, std::size_t &added, bool stealing, bool allow_stealing)

bool add_new (ThreadPool *receiver, std::size_t qidx, std::size_t &added, bool stealing, bool allow_stealing)

std::size_t get_new_tasks_queue_length () const

std::int64_t get_thread_count (thread_schedule_state state = thread_schedule_state::unknown, thread_priority priority = thread_priority::default_) const

void abort_all_suspended_threads ()

bool enumerate_threads (util::function_nonser<bool> thread_id_type > const &f, thread_schedule_state state) const

void increment_num_pending_misses (std::size_t = 1)

void increment_num_pending_accesses (std::size_t = 1)

void increment_num_stolen_from_pending (std::size_t = 1)

void increment_num_stolen_from_staged (std::size_t = 1)

void increment_num_stolen_to_pending (std::size_t = 1)

void increment_num_stolen_to_staged (std::size_t = 1)

bool dump_suspended_threads (std::size_t, std::int64_t&, bool)

void debug_info ()

void on_start_thread (std::size_t)
```
void on_stop_thread(std::size_t)
void on_error (std::size_t, std::exception_ptr const&)

Public Members

std::size_t num_queues_
std::size_t domain_
std::vector<ThreadQueue*> queues_

Defines

QUEUE_HOLDER_THREAD_DEBUG
namespace hpx

Functions

static hpx::debug::enable_print<QUEUE_HOLDER_THREAD_DEBUG> hpx::tq_deb("QH_THRD")
namespace threads

namespace policies

Enums

enum [anonymous]
Values:
max_thread_count = 1000
enum [anonymous]
Values:
round_robin_rollover = 1

Functions

std::size_t fast_mod (std::size_t const input, std::size_t const ceil)
template< typename QueueType>
struct queue_holder_thread
Public Types

template<>
using thread_holder_type = queue_holder_thread<QueueType>

template<>
using mutex_type = std::mutex

typedef std::unique_lock<mutex_type> scoped_lock

template<>
using thread_heap_type = std::list<thread_id_type, util::internal_allocator<thread_id_type>>

template<>
using task_description = thread_init_data

template<>
using thread_map_type = std::unordered_set<thread_id_type, std::hash<thread_id_type>, std::equal_to<thread_id_type>, util::internal_allocator<thread_id_type>>

template<>
using terminated_items_type = lockfree_fifo::apply<thread_data*>::type

Public Functions

queue_holder_thread(QueueType *bp_queue, QueueType *hp_queue, QueueType *np_queue, QueueType *lp_queue, std::size_t domain, std::size_t queue, std::size_t thread_num, std::size_t owner, const thread_queue_init_parameters &init)

~queue_holder_thread()

bool owns_bp_queue() const

bool owns_hp_queue() const

bool owns_np_queue() const

bool owns_lp_queue() const

std::size_t worker_next (std::size_t const workers) const

void schedule_thread(threads::thread_id_ref_type thrd, thread_priority priority, bool other_end = false)

bool cleanup_terminated(std::size_t thread_num, bool delete_all)

void create_thread(thread_init_data &data, thread_id_ref_type *tid, std::size_t thread_num, error_code &ec)

void create_thread_object (threads::thread_id_ref_type &tid, threads::thread_init_data &data)

void recycle_thread(thread_id_type tid)

void add_to_thread_map (threads::thread_id_type tid)

void remove_from_thread_map (threads::thread_id_type tid, bool dealloc)

bool get_next_thread_HP (threads::thread_id_ref_type &thrd, bool stealing, bool check_new)
bool get_next_thread (threads::thread_id_ref_type &thrd, bool stealing)
std::size_t add_new_HP (std::int64_t add_count, thread_holder_type *addfrom, bool stealing)
std::size_t add_new (std::int64_t add_count, thread_holder_type *addfrom, bool stealing)
std::size_t get_queue_length ()
std::size_t get_thread_count_staged (thread_priority priority) const
std::size_t get_thread_count_pending (thread_priority priority) const
std::size_t get_thread_count (thread_schedule_state state = thread_schedule_state::unknown, thread_priority priority = thread_priority::default_) const
void destroy_thread (threads::thread_data *thrd, std::size_t thread_num, bool xthread)
    Destroy the passed thread as it has been terminated.
void abort_all_suspended_threads ()
bool enumerate_threads (util::function_nonser<bool> &f, thread_id_type const &thid, thread_schedule_state state = thread_schedule_state::unknown const
void debug_info ()
void debug_queues (const char *prefix)

Public Members

QueueType *const bp_queue_
QueueType *const hp_queue_
QueueType *const np_queue_
QueueType *const lp_queue_
const std::size_t domain_index_
const std::size_t queue_index_
const std::size_t thread_num_
const std::size_t owner_mask_
util::cache_line_data<mutex_type> thread_map_mtx_
thread_heap_type thread_heap_small_
thread_heap_type thread_heap_medium_
thread_heap_type thread_heap_large_
thread_heap_type thread_heap_huge_
thread_heap_type thread_heap_nostack_
util::cache_line_data<std::tuple<std::size_t, std::size_t>> rollover_counters_
thread_map_type thread_map_
util::cache_line_data<std::atomic<std::int32_t>> thread_map_count_
terminated_items_type terminated_items_
util::cache_line_data<std::atomic<std::int32_t>> terminated_items_count_
thread_queue_init_parameters parameters_

Public Static Functions

static void deallocate (threads::thread_data *p)

Public Static Attributes

util::internal_allocator<threads::thread_data> thread_alloc_

struct queue_data_print

Public Functions

template<>
queue_data_print (const queue_holder_thread *q)

Public Members

template<>
const queue_holder_thread *q_

Friends

std::ostream &operator<< (std::ostream &os, const queue_data_print &d)

struct queue_mc_print

Public Functions

template<>
queue_mc_print (const QueueType *const q)

Public Members

template<>
const QueueType *const q_
Friends

\[ \text{std::ostream} & \text{operator}\ll (\text{std::ostream} & os, \text{const} \text{queue_mc_print} &d) \]

Defines

SHARED_PRIORITY_SCHEDULER_DEBUG
SHARED_PRIORITY_QUEUE_SCHEDULER_API

namespace hpx

Typedefs

using print_onoff = hpx::debug::enable_print<SHARED_PRIORITY_SCHEDULER_DEBUG>
using print_on = hpx::debug::enable_print<false>

Functions

static print_onoff hpx::spq_deb("SPQUEUE")
static print_on hpx::spq_arr("SPQUEUE")

namespace threads

namespace policies

Typedefs

using default_shared_priority_queue_scheduler_terminated_queue = lockfree_fifo

struct core_ratios

Public Functions

core_ratios (std::size_t high_priority, std::size_t normal_priority, std::size_t low_priority)

Public Members

std::size_t high_priority
std::size_t normal_priority
std::size_t low_priority

template<typename Mutex = std::mutex, typename PendingQueuing = concurrentqueue_fifo, typename Termin
class shared_priority_queue_scheduler: public scheduler_base
#include <shared_priority_queue_scheduler.hpp> The shared_priority_queue_scheduler maintains a set of high, normal, and low priority queues. For each priority level there is a core/queue ratio which determines how many cores share a single queue. If the high priority core/queue ratio is 4 the first 4 cores will share a single high priority queue, the next 4 will share another one and so on. In addition, the shared_priority_queue_scheduler is NUMA-aware and takes NUMA scheduling hints into account when creating and scheduling work.

Warning: PendingQueuing lifo causes lockup on termination

Public Types

template<>
using has_periodic_maintenance = std::false_type
template<>
using thread_queue_type = thread_queue_mc<Mutex, PendingQueuing, PendingQueuing, TerminatedQueuing>
template<>
using thread_holder_type = queue_holder_thread<thread_queue_type>
typedef init_parameter init_parameter_type

Public Functions

shared_priority_queue_scheduler (init_parameter const &init)
virtual ~shared_priority_queue_scheduler ()
void set_scheduler_mode (scheduler_mode mode)
void abort_all_suspended_threads ()
std::size_t local_thread_number ()
bool cleanup_terminated (bool delete_all)
bool cleanup_terminated (std::size_t, bool delete_all)
void create_thread (thread_init_data &data, thread_id_ref_type *thrd, error_code &ec)
template<typename T>
bool steal_by_function (std::size_t domain, std::size_t q_index, bool steal_numa, bool
steal_core, thread_holder_type *origin, T &var, const char
*prefix, util::function_nonser<bool> std::size_t, std::size_t,
thread_holder_type*, T&, bool, bool
> operation_HP, util::function_nonser<bool>std::size_t, std::size_t, thread_holder_type*, T&,
bool, bool> operation
virtual bool get_next_thread (std::size_t thread_num, bool running,
threads::thread_id_ref_type &thrd, bool enable_stealing)

Return the next thread to be executed, return false if none available.

virtual bool wait_or_add_new (std::size_t, bool, std::int64_t&, bool, std::size_t
&added)

Return the next thread to be executed, return false if none available.
void schedule_thread(threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback, thread_priority priority = thread_priority::normal)

Schedule the passed thread.

void schedule_thread_last(threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback, thread_priority priority = thread_priority::normal)

Put task on the back of the queue: not yet implemented just put it on the normal queue for now

void destroy_thread(threads::thread_data *thrd)

std::int64_t get_queue_length(std::size_t thread_num = std::size_t(-1)) const

std::int64_t get_thread_count(thread_schedule_state state = thread_schedule_state::unknown, thread_priority priority = thread_priority::default_, std::size_t thread_num = std::size_t(-1), bool = false) const

bool is_core_idle(std::size_t num_thread) const

bool enumerate_threads (util::function_nonser<bool>&f, thread_schedule_state state = thread_schedule_state::unknown const

void on_start_thread(std::size_t local_thread)

void on_stop_thread(std::size_t thread_num)

void on_error (std::size_t thread_num, std::exception_ptr const&)

Public Static Functions

static std::string get_scheduler_name ()

Protected Types

typedef queue_holder_numa<thread_queue_type> numa_queues

Protected Attributes

std::array<std::size_t, HPX_HAVE_MAX_NUMA_DOMAIN_COUNT> q_counts_
std::array<std::size_t, HPX_HAVE_MAX_NUMA_DOMAIN_COUNT> q_offset_
std::array<numa_queues, HPX_HAVE_MAX_NUMA_DOMAIN_COUNT> numa_holder_
std::vector<std::size_t> d_lookup_
std::vector<std::size_t> q_lookup_
core_ratios cores_per_queue_
bool round_robin_
bool steal_hp_first_
bool numa_stealing_
bool core_stealing_
std::size_t num_workers_
std::size_t num_domains_
detail::affinity_data const &affinity_data_
const thread_queue_init_parameters queue_parameters_
std::mutex init_mutex
bool initialized_
bool debug_init_
std::atomic<std::size_t> thread_init_counter_
std::size_t pool_index_

struct init_parameter

Public Functions

template<>
init_parameter (std::size_t num_worker_threads, const core_ratios &cores_per_queue, detail::affinity_data const &affinity_data, const thread_queue_init_parameters &thread_queue_init, char const *description = "shared_priority_queue_scheduler")

Public Members

template<>
std::size_t num_worker_threads_

template<>
core_ratios cores_per_queue_

template<>
thread_queue_init_parameters thread_queue_init_

template<>
detail::affinity_data const &affinity_data_

template<>
char const *description_

namespace hpx

namespace threads

namespace policies
Typedefs

using default_static_priority_queue_scheduler_terminated_queue = lockfree_fifo

template-typename Mutex = std::mutex, typename PendingQueuing = lockfree_fifo, typename StagedQueuing = lockfree_fifo, typename TerminatedQueuing = default_static_priority_queue_scheduler_terminated_queue
class static_priority_queue_scheduler : public hpx::threads::policies::local_priority_queue_scheduler<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing>
#include <static_priority_queue_scheduler.hpp> The static_priority_queue_scheduler maintains exactly one queue of work items (threads) per OS thread, where this OS thread pulls its next work from. Additionally it maintains separate queues: several for high priority threads and one for low priority threads. High priority threads are executed by the first N OS threads before any other work is executed. Low priority threads are executed by the last OS thread whenever no other work is available. This scheduler does not do any work stealing.

Public Types

template<>
using base_type = local_priority_queue_scheduler<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing>
template<>
using init_parameter_type = typename base_type::init_parameter_type

Public Functions

static_priority_queue_scheduler (init_parameter_type const &init, bool deferred_initialization = true)

void set_scheduler_mode (scheduler_mode mode)

Public Static Functions

static std::string get_scheduler_name ()

namespace hpx

namespace threads

namespace policies

Typedefs

using default_static_queue_scheduler_terminated_queue = lockfree_fifo

template-typename Mutex = std::mutex, typename PendingQueuing = lockfree_fifo, typename StagedQueuing = lockfree_fifo, typename TerminatedQueuing = default_static_queue_scheduler_terminated_queue
class static_queue_scheduler : public hpx::threads::policies::local_queue_scheduler<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing>
#include <static_queue_scheduler.hpp> The static_queue_scheduler maintains exactly one queue of work items (threads) per OS thread, where this OS thread pulls its next work from.
Public Types

typedef local_queue_scheduler<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing> base_type

Public Functions

static_queue_scheduler (typename base_type::init_parameter_type const &init,
bool deferred_initialization = true)

void set_scheduler_mode (scheduler_mode mode)

bool get_next_thread (std::size_t num_thread, bool, threads::thread_id_ref_type &thrd,
bool)
Return the next thread to be executed, return false if none is available

bool wait_or_add_new (std::size_t num_thread, bool running, std::int64_t &idle_loop_count, bool,
std::size_t &added)
This is a function which gets called periodically by the thread manager to allow for maintenance
tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to
be terminated (i.e. no more work has to be done).

Public Static Functions

static std::string get_scheduler_name ()

namespace hpx

namespace threads

namespace policies

template<typename Mutex, typename PendingQueuing, typename StagedQueuing, typename TerminatedQueuing>

class thread_queue

Public Functions

bool cleanup_terminated_locked (bool delete_all = false)
This function makes sure all threads which are marked for deletion (state is terminated) are
properly destroyed.
This returns ‘true’ if there are no more terminated threads waiting to be deleted.

bool cleanup_terminated (bool delete_all = false)

thread_queue (std::size_t queue_num = std::size_t(-1), thread_queue_init_parameters parameters = {})

~thread_queue ()

std::int64_t get_queue_length (std::memory_order order =
std::memory_order_acquire) const
```cpp
std::int64_t get_pending_queue_length (std::memory_order order = std::memory_order_acquire) const
std::int64_t get_staged_queue_length (std::memory_order order = std::memory_order_acquire) const

constexpr void increment_num_pending_misses (std::size_t = 1)
constexpr void increment_num_pending_accesses (std::size_t = 1)
constexpr void increment_num_stolen_from_pending (std::size_t = 1)
constexpr void increment_num_stolen_from_staged (std::size_t = 1)
constexpr void increment_num_stolen_to_pending (std::size_t = 1)
constexpr void increment_num_stolen_to_staged (std::size_t = 1)

void create_thread (thread_init_data &data, thread_id_ref_type *id, error_code &ec)
void move_work_items_from (thread_queue *src, std::int64_t count)
void move_task_items_from (thread_queue *src, std::int64_t count)
bool get_next_thread (threads::thread_id_ref_type &thrd, bool allow_stealing = false, bool steal = false)
  Return the next thread to be executed, return false if none is available
void schedule_thread (threads::thread_id_ref_type thrd, bool other_end = false)
  Schedule the passed thread.
void destroy_thread (threads::thread_data *thrd)
  Destroy the passed thread as it has been terminated.
std::int64_t get_thread_count (thread_schedule_state state = thread_schedule_state::unknown) const
  Return the number of existing threads with the given state.
void abort_all_suspended_threads ()
bool enumerate_threads (util::function_nonser<bool> thread_id_type
  > const &f, thread_schedule_state state = thread_schedule_state::unknown const
bool wait_or_add_new (bool, std::size_t &added, bool steal = false)
  This is a function which gets called periodically by the thread manager to allow for maintenance
tasks to be executed in the scheduler. Returns true if the OS thread calling this function has to
be terminated (i.e. no more work has to be done).
bool wait_or_add_new (bool running, std::size_t &added, thread_queue *addfrom, bool
  steal = false)
bool dump_suspended_threads (std::size_t num_thread, std::int64_t &idle_loop_count, bool
  running)
void on_start_thread (std::size_t)
void on_stop_thread (std::size_t)
void on_error (std::size_t, std::exception_ptr const &)
```

2.8. API reference 1125
Public Static Functions

static void deallocate (threads::thread_data *p)

Protected Functions

template<typename Lock>
void create_thread_object (threads::thread_id_ref_type &thrd, threads::thread_init_data &data, Lock &lk)

std::size_t add_new (std::int64_t add_count, thread_queue *addfrom, std::unique_lock<mutex_type> &lk, bool steal = false)

bool add_new_always (std::size_t &added, thread_queue *addfrom, std::unique_lock<mutex_type> &lk, bool steal = false)

void recycle_thread (thread_id_type thrd)

Protected Static Attributes

util::internal_allocator<typeid thread_queue<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing>>

Private Types

template<>
using mutex_type = Mutex
template<>
using thread_map_type = std::unordered_set<thread_id_type, std::hash<thread_id_type>, std::equal_to<thread_id_type>>
template<>
using thread_heap_type = std::vector<thread_id_type, util::internal_allocator<thread_id_type>>
template<>
using thread_description_ptr = thread_id_ref_type
template<>
using work_items_type = typename PendingQueuing::template apply<thread_description_ptr>::type
template<>
using task_items_type = typename StagedQueuing::template apply<task_description*>::type
template<>
using terminated_items_type = typename TerminatedQueuing::template apply<thread_data*>::type

Private Members

thread_queue_init_parameters parameters_
mutex_type mtx_
thread_map_type thread_map_
std::atomic<std::int64_t> thread_map_count_
work_items_type work_items_
terminated_items_type terminated_items_
\texttt{std::atomic<\textit{std::int64\_t}> terminated\_items\_count_}
\texttt{task\_items\_type new\_tasks_}
\texttt{thread\_heap\_type thread\_heap\_small_}
\texttt{thread\_heap\_type thread\_heap\_medium_}
\texttt{thread\_heap\_type thread\_heap\_large_}
\texttt{thread\_heap\_type thread\_heap\_huge_}
\texttt{thread\_heap\_type thread\_heap\_nostack_}
\texttt{util::cache\_line\_data<\textit{std::atomic<\textit{std::int64\_t}>}> new\_tasks\_count_}
\texttt{util::cache\_line\_data<\textit{std::atomic<\textit{std::int64\_t}>}> work\_items\_count_}

\textbf{Public Members}

\begin{verbatim}
template<> thread\_init\_data data
\end{verbatim}

\textbf{Defines}

\texttt{TH\_QUEUE\_MC\_DEBUG}

\texttt{namespace hpx}

\textbf{Functions}

\begin{verbatim}
static hpx::debug::enable\_print<\textit{TH\_QUEUE\_MC\_DEBUG}> hpx::tqmc\_deb("TQ\_MC")
\end{verbatim}

\texttt{namespace threads}

\texttt{namespace policies}

\begin{verbatim}
template<typename Mutex, typename PendingQueuing, typename StagedQueuing, typename TerminatedQueuing>
class thread\_queue\_mc
\end{verbatim}

\textbf{Public Types}

\begin{verbatim}
typedef Mutex mutex\_type
\end{verbatim}

\begin{verbatim}
using thread\_queue\_type = thread\_queue\_mc<Mutex, PendingQueuing, StagedQueuing, TerminatedQueuing>
\end{verbatim}

\begin{verbatim}
using thread\_heap\_type = std::list<thread\_id\_type, util::internal\_allocator<thread\_id\_type>>
\end{verbatim}

\begin{verbatim}
using task\_description = thread\_init\_data
\end{verbatim}

\begin{verbatim}
using thread\_description = thread\_data
\end{verbatim}
typedef PendingQueuing::template apply<thread_id_ref_type>::type work_items_type
typedef concurrentqueue_fifo::apply<task_description>::type task_items_type

Public Functions

std::size_t add_new (std::int64_t add_count, thread_queue_type *addfrom, bool stealing)

void set_holder (queue_holder_thread<thread_queue_type>*holder)

void create_thread (thread_init_data &data, thread_id_ref_type *id, error_code &ec)

bool get_next_thread (threads::thread_id_ref_type &thrd, bool other_end, bool check_new = false)
    Return the next thread to be executed, return false if none is available

void schedule_work (threads::thread_id_ref_type thrd, bool other_end)
    Schedule the passed thread (put it on the ready work queue)

void on_start_thread (std::size_t)

void on_stop_thread (std::size_t)

void on_error (std::size_t, std::exception_ptr const&)

Public Members

thread_queue_init_parameters parameters_

const int queue_index_

queue_holder_thread<thread_queue_type>*holder_

work_items_type work_items_

util::cache_line_data<std::atomic<std::int32_t>> new_tasks_count_

util::cache_line_data<std::atomic<std::int32_t>> work_items_count_
The contents of this module can be included with the header `hpx/modules/serialization.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/serialization.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
template<typename T>
struct serialize_non_intrusive<T, std::enable_if_t<has_serialize_adl<T>::value>>
```

**Public Static Functions**

```cpp
template<typename Archive>
static void call(Archive &ar, T &t, unsigned)
```

```cpp
namespace hpx

namespace serialization

class access

**Public Static Functions**

```cpp
template<typename Archive, typename T>
static void serialize(Archive &ar, T &t, unsigned)
```

```cpp
template<typename Archive, typename T>
static void save_base_object(Archive &ar, T const &t, unsigned)
```

```cpp
template<typename Archive, typename T>
static void load_base_object(Archive &ar, T &t, unsigned)
```

```cpp
template<typename T>
static std::string get_name(T const *t)
```

```cpp
template<typename T>
class has_serialize
```
## Public Static Attributes

```cpp
cconstexpr bool value = decltype(test<T>(0))::value
```

## Private Static Functions

```cpp
template<typename T1>
static std::false_type test(...) 

template<typename T1, typename = decltype(std::declval<typename std::remove_const<T1>::type&>().serialize(std::declval<hpx::serialization::output_archive&>(), 0u))>
static std::true_type test(int)
```

## Public Types

```cpp
template<>
using type = std::conditional_t<hpx::traits::is_intrusive_polymorphic_v<T>, intrusive_polymorphic, std::conditional_t<has_serialize<T>::value, intrusive_usual, std::conditional_t<std::is_empty<T>::value, empty, non_intrusive>>> 
```

## Public Static Functions

### struct empty

```cpp
template<class Archive>
static void call(Archive&, T&, unsigned)
```

### Public Static Functions

```cpp
template<>
static void call(hpx::serialization::input_archive &ar, T &t, unsigned)
```

### struct intrusive_polymorphic

```cpp
template<>
static void call(hpx::serialization::output_archive &ar, T const &t, unsigned)
```

### Public Static Functions

```cpp
template<class Archive>
static void call(Archive &ar, T &t, unsigned)
```

### struct non_intrusive

```cpp
template<class Archive>
static void call(Archive &ar, T &t, unsigned)
```
Public Static Functions

```cpp
template<class Archive>
static void call (Archive & ar, T & t, unsigned)
```

template< typename T>
class has_serialization

Public Static Attributes

```cpp
constexpr bool value = decltype(test<T>(0))::value
```

Private Static Functions

```cpp
template< typename T1>
static std::false_type test ( ... )
```

```cpp
template< typename T1, typename = decltype(serialize(std::declval<hpx::serialization::output_archive&>(), std::declval<T1&>(), 0u))>
static std::true_type test ( int )
```

```cpp
template< typename T>
class has_struct_serialization
```

Public Static Functions

```cpp
template< typename T1>
static std::false_type test ( ... )
```

```cpp
template< typename T1, typename = decltype(serialize_struct(std::declval<hpx::serialization::output_archive&>(), std::declval<T1&>(), 0u, hpx::traits::detail::arity<T1>()))>
static std::true_type test ( int )
```

Public Static Attributes

```cpp
constexpr bool value = decltype(test<T>(0))::value
```

```cpp
template< typename T>
struct serialize_brace_initialized<T, std::enable_if_t<has_struct_serialization<T>::value>>
```

Public Static Functions

```cpp
template< typename Archive>
static void call (Archive & ar, T & t, unsigned)
```

```cpp
template< typename T>
struct serialize_non_intrusive<T, std::enable_if_t<has_serialization<T>::value>>
```
Public Static Functions

template<typename Archive>
static void call (Archive & ar, T & t, unsigned)

namespace hpx

namespace serialization

Functions

template<typename T>
array<T> make_array (T * begin, std::size_t size)

template<typename Archive, typename T, std::size_t N>
void serialize (Archive & ar, std::array<T, N> & a, const unsigned int)

template<typename T>
output_archive & operator<<(output_archive & ar, array<T> t)

template<typename T>
input_archive & operator>>(input_archive & ar, array<T> t)

template<typename T>
output_archive & operator&(output_archive & ar, array<T> t)

template<typename T>
input_archive & operator&(input_archive & ar, array<T> t)

template<typename T, std::size_t N>
output_archive & operator<<(output_archive & ar, T (& t)[N])

template<typename T, std::size_t N>
input_archive & operator>>(input_archive & ar, T (& t)[N])

template<typename T, std::size_t N>
output_archive & operator&(output_archive & ar, T (& t)[N])

template<typename T, std::size_t N>
input_archive & operator&(input_archive & ar, T (& t)[N])

template<typename T>
class array

Public Types

template<>
using value_type = T
### Public Functions

**array** (value_type *, std::size_t)

value_type *`address` ( ) const

std::size_t`count` ( ) const

template<class Archive>
void serialize_optimized (Archive &ar, unsigned int, std::false_type)

void serialize_optimized (output_archive &ar, unsigned int, std::true_type)

void serialize_optimized (input_archive &ar, unsigned int, std::true_type)

template<class Archive>
void serialize (Archive &ar, unsigned int v)

### Private Members

value_type *`m_t`

std::size_t`m_element_count`

template<typename Derived, typename Base>

struct base_object_type<Derived, Base, std::true_type>

### Public Functions

**base_object_type** (Derived &d)

template<class Archive>
void save (Archive &ar, unsigned) const

template<class Archive>
void load (Archive &ar, unsigned)

HPX_SERIALIZATION_SPLIT_MEMBER ()

### Public Members

Derived &d_

namespace hpx

namespace serialization
Functions

template<typename Base, typename Derived>
base_object_type<Derived, Base> base_object (Derived &d)

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)

structure base_object_type

Public Functions

    base_object_type (Derived &d)

    template<typename Archive>
    void serialize (Archive &ar, unsigned)

Public Members

    Derived &d_

structure base_object_type<Derived, Base, std::true_type>

Public Functions

    base_object_type (Derived &d)

    template<class Archive>
    void save (Archive &ar, unsigned) const

    template<class Archive>
    void load (Archive &ar, unsigned)

    HPX_SERIALIZATION_SPLIT_MEMBER()
Public Members

Derived &d_

namespace hpx

namespace serialization

Enums

enum archive_flags

Values:

no_archive_flags = 0x00000000
enable_compression = 0x00002000
endian_big = 0x00004000
endian_little = 0x00008000
disable_array_optimization = 0x00010000
disable_data_chunking = 0x00020000
all_archive_flags = 0x0003e000

Functions

void reverse_bytes (std::size_t size, char *address)
template<typename Archive>
void save_binary (Archive &ar, void const *address, std::size_t count)
template<typename Archive>
void load_binary (Archive &ar, void *address, std::size_t count)
template<typename Archive>
std::size_t current_pos (const Archive &ar)
template<typename Archive>
struct basic_archive

Public Functions

virtual ~basic_archive ()
template<typename T>
void invoke (T &t)
bool enable_compression () const
bool endian_big () const
bool endian_little () const
bool disable_array_optimization () const
bool disable_data_chunking() const

std::uint32_t flags() const

bool is_preprocessing() const

std::size_t current_pos() const

void save_binary(void const *address, std::size_t count)

void load_binary(void *address, std::size_t count)

void reset()

template<typename T>
T &get_extra_data()

template<typename T>
T *try_get_extra_data()

Public Static Attributes

const std::uint64_t npos = std::uint64_t(-1)

Protected Functions

basic_archive(std::uint32_t flags)

basic_archive(basic_archive const&)

basic_archive &operator=(basic_archive const&)

Protected Attributes

std::uint32_t flags_

std::size_t size_

detail::extra_archive_data extra_data_

namespace hpx

namespace serialization

struct binary_filter
Public Functions

```cpp
virtual void set_max_length (std::size_t size) = 0
virtual void save (void const *src, std::size_t src_count) = 0
virtual bool flush (void *dst, std::size_t dst_count, std::size_t &written) = 0
virtual std::size_t init_data (char const *buffer, std::size_t size, std::size_t buffer_size) = 0
virtual void load (void *dst, std::size_t dst_count) = 0

template<class T>
void serialize (T &, unsigned)

HPX_SERIALIZATION_POLYMORPHIC_ABSTRACT (binary_filter)

virtual ~binary_filter ()
```

namespace hpx

namespace serialization

Functions

```cpp
template<std::size_t N>
void serialize (input_archive &ar, std::bitset<N> &d, unsigned)

template<std::size_t N>
void serialize (output_archive &ar, std::bitset<N> const &bs, unsigned)
```

namespace hpx

namespace serialization

Functions

```cpp
template<
typename Archive, typename T>
void serialize_struct (Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<0>)

template<
typename Archive, typename T>
void serialize_struct (Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<1>)

template<
typename Archive, typename T>
void serialize_struct (Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<2>)

template<
typename Archive, typename T>
void serialize_struct (Archive &archive, T &t, const unsigned int version,
                      hpx::traits::detail::size<3>)
```

2.8. API reference
namespace serialization
**Functions**

```cpp
template<typename T>
void serialize (input_archive &ar, std::complex<T> &c, unsigned)

namespace hpx

namespace serialization

struct erased_input_container
    Subclassed by hpx::serialization::input_container< Container >

Public Functions

virtual ~erased_input_container ()
virtual bool is_preprocessing () const
virtual void set_filter (binary_filter *filter) = 0
virtual void load_binary (void *address, std::size_t count) = 0
virtual void load_binary_chunk (void *address, std::size_t count) = 0

struct erased_output_container
    Subclassed by hpx::serialization::output_container< Container, Chunker >

Public Functions

virtual ~erased_output_container ()
virtual bool is_preprocessing () const
virtual void set_filter (binary_filter *filter) = 0
virtual void save_binary (void const *address, std::size_t count) = 0
virtual std::size_t save_binary_chunk (void const *address, std::size_t count) = 0
virtual void reset () = 0
virtual std::size_t get_num_chunks () const = 0
virtual void flush () = 0

namespace hpx

namespace serialization
```

2.8. API reference 1139
Functions

template<typename T, typename Allocator>
void serialize (input_archive &ar, std::deque<T, Allocator> &d, unsigned)

namespace hp

namespace serialization

Functions

template<typename Archive>
void save (Archive &ar, std::exception_ptr const &e, unsigned int)

namespace util

Enums

enum exception_type

Values:

unknown_exception = 0
std_runtime_error = 1
std_invalid_argument = 2
std_out_of_range = 3
std_logic_error = 4
std_bad_alloc = 5
std_bad_cast = 6
std_bad_typeid = 7
std_bad_exception = 8
std_exception = 9
boost_system_error = 10
hpx_exception = 11
hpx_thread_interrupted_exception = 12
std_system_error = 14

namespace hp

namespace serialization
struct input_archive : public hpx::serialization::basic_archive<input_archive>

Public Types

using base_type = basic_archive<input_archive>

Public Functions

template<typename Container>
input_archive(Container &buffer, std::size_t inbound_data_size = 0, const std::vector<serialization_chunk> *chunks = nullptr)

template<typename T>
void invoke_impl(T &t)

template<typename T>
std::enable_if_t<!std::is_integral_v<T> && !std::is_enum_v<T>> load(T &t)

template<typename T>
std::enable_if_t<std::is_integral_v<T> || std::is_enum_v<T>> load(T &t)

void load(float &f)

void load(double &d)

void load(char &c)

void load(bool &b)

std::size_t bytes_read() const

std::size_t current_pos() const

Private Functions

template<typename T>
void load_bitwise(T &t, std::false_type)

template<typename T>
void load_bitwise(T &t, std::true_type)

template<class T>
void load_nonintrusively_polymorphic(T &t, std::false_type)

template<class T>
void load_nonintrusively_polymorphic(T &t, std::true_type)

template<typename T>
void load_integral(T &val, std::false_type)

template<typename T>
void load_integral(T &val, std::true_type)

template<class Promoted>
void load_integral_impl(Promoted &l)
void load_binary (void *address, std::size_t count)
void load_binary_chunk (void *address, std::size_t count)

Private Members

std::unique_ptr<erased_input_container> buffer_

Friends

friend hpx::serialization::basic_archive< input_archive >
friend hpx::serialization::array

namespace hpx

namespace serialization

template< typename Container >
struct input_container : public hpx::serialization::erased_input_container

Public Functions

input_container (Container const &cont, std::size_t inbound_data_size)
input_container (Container const &cont, std::vector<serialization_chunk> const *chunks, std::size_t inbound_data_size)
void set_filter (binary_filter *filter)
void load_binary (void *address, std::size_t count)
void load_binary_chunk (void *address, std::size_t count)

Public Members

Container const &cont_
std::size_t current_
std::unique_ptr<binary_filter> filter_
std::size_t decompressed_size_
std::vector<serialization_chunk> const *chunks_
std::size_t current_chunk_
std::size_t current_chunk_size_
Private Types

template<>
using access_traits = traits::serialization_access_data<Container>

Private Functions

std::size_t get_chunk_size (std::size_t chunk) const
std::uint8_t get_chunk_type (std::size_t chunk) const
chunk_data get_chunk_data (std::size_t chunk) const
std::size_t get_num_chunks () const

namespace hpx

namespace serialization

Functions

template<typename T, typename Allocator>
void serialize (input_archive &ar, std::list<T, Allocator> &ls, unsigned)

template<typename T, typename Allocator>
void serialize (output_archive &ar, const std::list<T, Allocator> &ls, unsigned)

namespace hpx

namespace serialization

Functions

template<typename Key, typename Value>
void serialize (input_archive &ar, std::pair<Key, Value> &t, unsigned)

template<typename Key, typename Value>
void serialize (output_archive &ar, const std::pair<Key, Value> &t, unsigned)

template<typename Key, typename Value, typename Comp, typename Alloc>
void serialize (input_archive &ar, std::map<Key, Value, Comp, Alloc> &t, unsigned)

template<typename Key, typename Value, typename Comp, typename Alloc>
void serialize (output_archive &ar, std::map<Key, Value, Comp, Alloc> const &t, unsigned)

namespace hpx

namespace serialization
Functions

```cpp
template<typename T>
void save (output_archive &ar, hpx::util::optional<T> const &o, unsigned);

template<typename T>
void load (input_archive &ar, hpx::util::optional<T> &o, unsigned);
```

```cpp
namespace hpx

namespace serialization

struct output_archive : public hpx::serialization::basic_archive<output_archive>

Public Types

```cpp
using base_type = basic_archive<output_archive>
```

Public Functions

```cpp
template<typename Container>
output_archive (Container &buffer, std::uint32_t flags = 0U,
                 std::vector<serialization_chunk> *chunks = nullptr,
                 binary_filter *filter = nullptr);

std::size_t bytes_written () const

std::size_t get_num_chunks () const

std::size_t current_pos () const

void reset ()

void flush ()

bool is_preprocessing () const
```

Protected Functions

```cpp
template<typename T>
void invoke_impl (T const &t)

template<typename T>
std::enable_if_t<!std::is_integral_v<T> && !std::is_enum_v<T>> save (T const &t)

template<typename T>
std::enable_if_t<!std::is_integral_v<T> || std::is_enum_v<T>> save (T t)

void save (float f)

void save (double d)

void save (char c)
```
void save (bool b)

template<typename T>
void save_bitwise (T const & t, std::false_type)

template<typename T>
void save_bitwise (T const & t, std::true_type)

template<typename T>
void save_nonintrusively_polymorphic (T const & t, std::false_type)

template<typename T>
void save_nonintrusively_polymorphic (T const & t, std::true_type)

template<typename T>
void save_integral (T val, std::false_type)

template<typename T>
void save_integral (T val, std::true_type)

template<class Promoted>
void save_integral_impl (Promoted l)

void save_binary (void const * address, std::size_t count)

void save_binary_chunk (void const * address, std::size_t count)

Protected Attributes

std::unique_ptr<erased_output_container> buffer_

Private Static Functions

static std::uint32_t make_flags (std::uint32_t flags, std::vector<serialization_chunk> *chunks)

Friends

friend hpx::serialization::basic_archive< output_archive >
friend hpx::serialization::array

namespace hpx

namespace serialization

template<typename Container, typename Chunker>
struct filtered_output_container : public hpx::serialization::output_container<Container, Chunker>
Public Types

template<>
using access_traits = traits::serialization_access_data<Container>

template<>
using base_type = output_container<Container, Chunker>

Public Functions

filtered_output_container (Container &cont, std::vector<serialization_chunk> *chunks = nullptr)

~filtered_output_container ()

void flush ()

void set_filter (binary_filter *filter)

void save_binary (void const *address, std::size_t count)

std::size_t save_binary_chunk (void const *address, std::size_t count)

Protected Attributes

std::size_t start_compressing_at_

binary_filter *filter_

template<typename Container, typename Chunker>
struct output_container : public hpx::serialization::erased_output_container

Subclassed by hpx::serialization::filtered_output_container< Container, Chunker >

Public Types

template<>
using access_traits = traits::serialization_access_data<Container>

Public Functions

output_container (Container &cont, std::vector<serialization_chunk> *chunks = nullptr)

~output_container ()

void flush ()

std::size_t get_num_chunks () const

void reset ()

void set_filter (binary_filter*)

void save_binary (void const *address, std::size_t count)

std::size_t save_binary_chunk (void const *address, std::size_t count)

bool is_preprocessing () const
Protected Attributes

Container &cont_

std::size_t current_

Chunker chunker_

template<typename IArch, typename OArch, typename Char>
class basic_any<IArch, OArch, Char, std::true_type>

Public Functions

constexpr basic_any ()
basic_any (basic_any const &x)
basic_any (basic_any &&x)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any (T &&x, typename std::enable_if<!std::is_copy_constructible<typename std::decay<T>::type>::value>::value* = nullptr)

template<typename T, typename ...Ts, typename Enable = typename std::enable_if<!std::is_constructible<typename std::decay<T>::type, Ts...>::value &&std::is_copy_constructible<typename std::decay<T>::type>::value>::type>
basic_any (std::in_place_type_t<T>, Ts&&... ts)

~basic_any ()
basic_any &operator= (basic_any const &x)
basic_any &operator= (basic_any &&rhs)

template<typename T, typename Enable = typename std::enable_if<!std::is_same<basic_any, typename std::decay<T>::type>::value>::type>
basic_any &operator= (T &&rhs)
basic_any &swap (basic_any &x)

std::type_info const &type () const

template<typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const &rhs) const
Private Functions

basic_any &assign(basic_any const &x)
void load(IArch &ar, const unsigned version)
void save(OArch &ar, const unsigned version) const

HPX_SERIALIZATION_SPLIT_MEMBER()

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 void new_object(void *object, std::true_type, Ts&&... ts)
template<typename T, typename ...Ts>
static void new_object(void *object, std::false_type, Ts&&... ts)

Friends

friend hpx::serialization::access

namespace hpx

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)

namespace util

Typedefs

using instead = basic_any<serialization::input_archive, serialization::output_archive, char, std::true_type>
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 T, typename Char>
class basic_any

Public Functions

constexpr basic_any ()

basic_any (basic_any const &x)

basic_any (basic_any &x)

template<
typename T, typename Ena = std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value>::type
basic_any (T &&x, typename std::enable_if<!std::is_copy_constructible<typename std::decay<T>::value>::value>::type* = nullptr)

template<
typename T, typename ...Ts, typename Ena = std::enable_if<!std::is_constructible<typename std::decay<T>::value::type>::value &&
std::is_copy_constructible<typename std::decay<T>::value>::value>::type
basic_any (std::in_place_type_t<T>, Ts&&... ts)

~basic_any ()

basic_any &operator= (basic_any const &x)

basic_any &operator= (basic_any &rhs)

template<
typename T, typename Ena = std::enable_if<!std::is_same<basic_any, typename std::decay<T>::value>::value &&
std::is_copy_constructible<typename std::decay<T>::value>::value>::type
basic_any &operator= (T &&rhs)

basic_any &swap (basic_any &x)

std::type_info const &type () const

template<
typename T>
T const &cast () const

bool has_value () const

void reset ()

bool equal_to (basic_any const &rhs) const
Private Functions

basic_any &\textbf{assign}(basic_any \textbf{const} \&x)

void \textbf{load}(IArch &\textit{ar}, \textbf{const} unsigned \textit{version})

void \textbf{save}(OArch &\textit{ar}, \textbf{const} unsigned \textit{version}) \textbf{const}

\texttt{HPX\_SERIALIZATION\_SPLIT\_MEMBER()}

Private Members

detail::any::fxn_ptr_table<IArch, OArch, Char, std::true_type> *\texttt{table}

void *\texttt{object}

Private Static Functions

\texttt{template<typename T, typename ...Ts> static} void \textbf{new\_object}(void *\&\textit{object}, std::true_type, Ts&&... ts)

\texttt{template<typename T, typename ...Ts> static} void \textbf{new\_object}(void *\&\textit{object}, std::false_type, Ts&&... ts)

Friends

friend hpx::util::hpx::serialization::access

\texttt{struct hash\_any}

Public Functions

\texttt{template<typename Char>} std::size_t \textbf{operator()}(\textbf{const} basic_any<serialization::input\_archive, serialization::output\_archive, Char, std::true_type &\textit{elem}) \textbf{const}

namespace hpx

namespace serialization

Enums

\texttt{enum chunk\_type}

\texttt{Values:}

\texttt{chunk\_type\_index} = 0

\texttt{chunk\_type\_pointer} = 1
Functions

`serialization_chunk create_index_chunk (std::size_t index, std::size_t size)`

`serialization_chunk create_pointer_chunk (void const *pos, std::size_t size, std::uint64_t rkey = 0)`

union chunk_data

Public Members

`std::size_t index_`

`void const *cpos_`

`void *pos_`

struct serialization_chunk

Public Members

`chunk_data data_`

`std::size_t size_`

`std::uint64_t rkey_`

`std::uint8_t type_`

Defines

`HPX_SERIALIZATION_SPLIT_MEMBER()`

`HPX_SERIALIZATION_SPLIT_FREE (T)`

`HPX_SERIALIZATION_SPLIT_FREE_TEMPLATE (TEMPLATE, ARGS)`

namespace hpx

namespace serialization

Functions

template<typename T>
output_archive &operator<< (output_archive &ar, T const &t)

template<typename T>
input_archive &operator>> (input_archive &ar, T &t)

template<typename T>
output_archive &operator& (output_archive &ar, T const &t)

template<typename T>
input_archive &operator& (input_archive &ar, T &t)
namespace hpx

namespace serialization

template<typename T, typename Allocator = std::allocator<T>>
class serialize_buffer

Public Types

enum init_mode
Values:
  copy = 0
  reference = 1
  take = 2

template<>
using value_type = T

Public Functions

serialize_buffer (allocator_type const & alloc = allocator_type())
serialize_buffer (std::size_t size, allocator_type const & alloc = allocator_type())
serialize_buffer (T *data, std::size_t size, init_mode mode = copy, allocator_type const & alloc = allocator_type())

template<Deallocator>
serialize_buffer (T *data, std::size_t size, allocator_type const & alloc, Deallocator const & dealloc)

template<Deleter>
serialize_buffer (T *data, std::size_t size, init_mode mode, Deleter const & deleter, allocator_type const & alloc = allocator_type())

template<Deleter, Deallocator>
serialize_buffer (T *data, std::size_t size, allocator_type const & alloc, Deallocator const & dealloc, Deleter const & deleter)

serialize_buffer (T const *data, std::size_t size, allocator_type const & alloc = allocator_type())

serialize_buffer (T const *data, std::size_t size, Deleter const & deleter, allocator_type const & alloc = allocator_type())

serialize_buffer (T const *data, std::size_t size, init_mode mode, allocator_type const & alloc = allocator_type())

T *data ()
T const *data() const
T *begin()
T *end()
T &operator[](std::size_t idx)
T operator[](std::size_t idx) const
buffer_type data_array() const
std::size_t size() const
void resize_norealloc(std::size_t newsize)

Private Types

template<>
using allocator_type = Allocator
template<>
using buffer_type = boost::shared_array<T>

Private Functions

template<typename Archive>
void save(Archive &ar, unsigned int const) const

template<typename Archive>
void load(Archive &ar, unsigned int const)

Private Members

buffer_type data_
std::size_t size_
Allocator alloc_

Private Static Functions

static void no_deleter(T*)

template<typename Deallocator>
static void deleter(T*p, Deallocator dealloc, std::size_t size)
Friends

friend hpx::serialization::hpx::serialization::access

bool operator==(serialize_buffer const &rhs, serialize_buffer const &lhs)

namespace hpx

namespace serialization

Functions

template<typename T, typename Compare, typename Allocator>
void serialize(input_archive &ar, std::set<T, Compare, Allocator> &set, unsigned)

template<typename T, typename Compare, typename Allocator>
void serialize(output_archive &ar, std::set<T, Compare, Allocator> const &set, unsigned)

namespace hpx

namespace serialization

Functions

template<typename T>
void load(input_archive &ar, std::shared_ptr<T> &ptr, unsigned)

template<typename T>
void save(output_archive &ar, std::shared_ptr<T> const &ptr, unsigned)

namespace hpx

namespace serialization

Functions

template<typename Archive, typename ...Ts>
void serialize(Archive &ar, std::tuple<Ts...> &t, unsigned int version)

template<typename Archive>
void serialize(Archive&, std::tuple<>&, unsigned int)

namespace hpx

namespace serialization
Functions

template<typename Char, typename CharTraits, typename Allocator>
void serialize(input_archive & ar, std::basic_string<Char, CharTraits, Allocator> & s, unsigned)

template<typename Char, typename CharTraits, typename Allocator>
void serialize(output_archive & ar, std::basic_string<Char, CharTraits, Allocator> const & s, unsigned)

namespace hpx

namespace serialization

Functions

template<typename Archive, typename ...Ts>
void serialize(Archive & ar, hpx::tuple<Ts...> & t, unsigned int version)

template<typename Archive>
void serialize(Archive&, hpx::tuple<> &, unsigned)

template<typename Archive, typename ...Ts>
void load_construct_data(Archive & ar, hpx::tuple<Ts...> * t, unsigned int version)

template<typename Archive, typename ...Ts>
void save_construct_data(Archive & ar, hpx::tuple<Ts...> const * t, unsigned int version)

namespace hpx

namespace serialization

Functions

template<typename T>
void load(input_archive & ar, std::unique_ptr<T> & ptr, unsigned)

template<typename T>
void save(output_archive & ar, const std::unique_ptr<T> & ptr, unsigned)

namespace hpx

namespace serialization
Functions

template<typename Key, typename Value, typename Hash, typename KeyEqual, typename Alloc>
void serialize (input_archive & ar, std::unordered_map<Key, Value, Hash, KeyEqual, Alloc> & t, unsigned)

namespace hpx

namespace serialization

Functions

template<typename T>
void serialize (input_archive & ar, std::valarray<T> & arr, int)

template<typename T>
void serialize (output_archive & ar, std::valarray<T> const & arr, int)

namespace hpx

namespace serialization

Functions

template<typename ...Ts>
void save (output_archive & ar, std::variant<Ts...> const & v, unsigned)

template<typename ...Ts>
void load (input_archive & ar, std::variant<Ts...> & v, unsigned)

namespace hpx

namespace serialization

Functions

template<typename Allocator>
void serialize (input_archive & ar, std::vector<bool, Allocator> & v, unsigned)

template<typename T, typename Allocator>
void serialize (input_archive & ar, std::vector<T, Allocator> & v, unsigned)

template<typename Allocator>
void serialize (output_archive & ar, std::vector<bool, Allocator> const & v, unsigned)

template<typename T, typename Allocator>
void serialize (output_archive & ar, std::vector<T, Allocator> const & v, unsigned)
Defines

`MAKE_ARITY_FUNC`(*count*)

namespace hpx

    namespace traits

    Functions

    template<typename T, std::size_t N>
    constexpr auto is_brace_constructible() { return hpx::traits::static_cast<T*>(nullptr); }

    template<typename T, std::size_t N>
    constexpr auto is_paren_constructible() { return true; }

Defines

`HPX_IS_BITWISE_SERIALIZABLE`(*T*)

namespace hpx

    namespace traits

    Variables

    template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_bitwise_serializable_v = is_bitwise_serializable<T>::value;

Defines

`HPX_IS_NOT_BITWISE_SERIALIZABLE`(*T*)

namespace hpx

    namespace traits

    Variables

    template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_not_bitwise_serializable_v = true;
HPX Documentation, master

Defines

HPX_TRAITS_NONINTRUSIVE_POLYMORPHIC (Class)
HPX_TRAITS_NONINTRUSIVE_POLYMORPHIC_TEMPLATE (TEMPLATE, ARG_LIST)
HPX_TRAITS_SERIALIZED_WITH_ID (Class)
HPX_TRAITS_SERIALIZED_WITH_ID_TEMPLATE (TEMPLATE, ARG_LIST)

namespace hpx

namespace traits

Variables

template<typename T>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_intrusive_polymorphic_v=is_intrusive_polymorphic<T>::value

template<typename T>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_nonintrusive_polymorphic_v=is_nonintrusive_polymorphic<T>::value

template<typename T>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::traits::is_serialized_with_id_v=is_serialized_with_id<T>::value

namespace hpx

namespace traits

template<typename Container>
struct default_serialization_access_data
  Subclassed by hpx::traits::serialization_access_data< Container >

Public Types

template<>
using preprocessing_only = std::false_type

Public Static Functions

static constexpr bool is_preprocessing()

static constexpr void write (Container&, std::size_t, std::size_t, void const*)

static bool flush (serialization::binary_filter*, Container&, std::size_t, std::size_t size, std::size_t &written)

static constexpr void read (Container const&, std::size_t, std::size_t, void*)

static constexpr std::size_t init_data (Container const&,
  serialization::binary_filter*, std::size_t, std::size_t decompressed_size)

static constexpr void reset (Container&)

template<typename Container>
struct serialization_access_data : public hpx::traits::default_serialization_access_data<Container>
  Subclassed by hpx::traits::serialization_access_data< Container const >
### Public Static Functions

```cpp
static std::size_t size (Container const &cont)
static void resize (Container &cont, std::size_t count)
static void write (Container &cont, std::size_t count, std::size_t current, void *address)
static bool flush (serialization::binary_filter *filter, Container &cont, std::size_t current, std::size_t size, std::size_t &written)
static void read (Container const &cont, std::size_t count, std::size_t current, void *address)
static std::size_t init_data (Container const &cont, serialization::binary_filter *filter, std::size_t current, std::size_t decompressed_size)
```

### static_reinit

The contents of this module can be included with the header `hpx/modules/static_reinit.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/static_reinit.hpp`, not the particular header in which the functionality you would like to use is defined. See **Public API** for a list of names that are part of the public **HPX** API.

### Defines

```cpp
HPX_CORE_EXPORT_REINITIALIZABLE_STATIC
namespace hpx

namespace util

```

### Variables

```cpp
template<typename T, typename Tag = T, std::size_t N = 1>
struct HPX_CORE_EXPORT_REINITIALIZABLE_STATIC reinitializable_static
```

```cpp
template<typename T, typename Tag, std::size_t N>
struct reinitializable_static
```

### Public Types

```cpp
typedef T value_type
typedef T &reference
typedef T const &const_reference
```
Public Functions

HPX_NON_COPYABLE (reinitializable_static)

reinitializable_static()

template<typename U>
reinitializable_static(U const &val)

operator reference()

operator const_reference() const

reference get (std::size_t item = 0)

const_reference get (std::size_t item = 0) const

Private Types

typedef std::add_pointer<value_type>::type pointer

typedef std::aligned_storage<sizeof(value_type), std::alignment_of<value_type>::value>::type storage_type

Private Static Functions

static void default_construct ()

template<typename U>
static void value_construct (U const &v)

static void destruct ()

static void default_constructor ()

template<typename U>
static void value_constructor (U const *pv)

static pointer get_address (std::size_t item)

Private Static Attributes

reinitializable_static<T, Tag, N>::storage_type data_

std::once_flag constructed_

namespace hpx

namespace util
Functions

void **reinit_register** (util::function_nonser<void> > const &constructutil::function_nonser<void> const &destruct
void **reinit_construct** ()
void **reinit_destruct** ()

string_util

The contents of this module can be included with the header hpx/modules/string_util.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/string_util.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace string_util

Functions

template<typename CharT, class Traits, class Alloc>
void to_lower (std::basic_string<CharT, Traits, Alloc> &s)

namespace hpx

namespace string_util

Functions

template<typename CharT, typename Traits, typename Allocator>
detail::is_any_of_pred<CharT, Traits, Allocator> **is.any.of** (std::basic_string<CharT, Traits, Allocator> const &chars)
auto **is.any.of** (char const *chars)

struct is_space

Public Functions

bool operator() (int c) const

namespace hpx

namespace string_util

2.8. API reference 1161
Enums

enum token_compress_mode
  Values:
  off
  on

Functions

template<typename Container, typename Predicate, typename CharT, typename Traits, typename Allocator>
void split(Container &container, std::basic_string<CharT, Traits, Allocator> const &str, Predicate &&pred, token_compress_mode compress_mode = token_compress_mode::off)

template<typename Container, typename Predicate>
void split(Container &container, char const *str, Predicate &&pred, token_compress_mode compress_mode = token_compress_mode::off)

namespace hpx

namespace string_util

Functions

template<typename CharT, class Traits, class Alloc>
void trim(std::basic_string<CharT, Traits, Alloc> &s)

template<typename CharT, class Traits, class Alloc>
std::basic_string<CharT, Traits, Alloc> trim_copy(std::basic_string<CharT, Traits, Alloc> const &s)

synchronization

The contents of this module can be included with the header hpx/modules/synchronization.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/synchronization.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace experimental

template<typename ReadWriteT, typename ReadT, typename Allocator>
class async_rw_mutex
  #include <async_rw_mutex.hpp> Read-write mutex where access is granted to a value through senders.

The wrapped value is accessed through read and readwrite, both of which return senders which call set_value on a connected receiver when the wrapped value is safe to read or write. The senders send the value through a wrapper type which is implicitly convertible to a reference of the wrapped value. Read-only senders send wrappers that are convertible to const references.
A read-write sender gives exclusive access to the wrapped value, while a read-only sender gives shared (with other read-only senders) access to the value.

A void mutex acts as a mutex around some user-managed resource, i.e. the void mutex does not manage any value and the types sent by the senders are not convertible. The sent types are copyable and release access to the protected resource when released.

The order in which senders call set_value is determined by the order in which the senders are retrieved from the mutex. Connecting and starting the senders is thread-safe.

Retrieving senders from the mutex is not thread-safe.

The mutex is movable and non-copyable.

**Public Types**

```cpp
template<>
using read_type = std::decay_t<ReadT> const
template<>
using readwrite_type = std::decay_t<ReadWriteT>
template<>
using value_type = readwrite_type
template<>
using read_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::async_rw_mutex_access_type::read>
template<>
using readwrite_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::async_rw_mutex_access_type::readwrite>
template<>
using allocator_type = Allocator
```

**Public Functions**

```cpp
async_rw_mutex ()
```

```cpp
template<typename U, typename = std::enable_if_t<!std::is_same<std::decay_t<U>, async_rw_mutex>::value>>
async_rw_mutex (U &&u, allocator_type const &alloc = {})
```

```cpp
async_rw_mutex (async_rw_mutex&&)
```

```cpp
async_rw_mutex &operator= (async_rw_mutex&&)
```

```cpp
async_rw_mutex (async_rw_mutex const&)
```

```cpp
async_rw_mutex &operator= (async_rw_mutex const&)
```

```cpp
sender<detail::async_rw_mutex_access_type::read> read ()
```

```cpp
sender<detail::async_rw_mutex_access_type::readwrite> readwrite ()
```
**Private Types**

```cpp
template<>
using shared_state_type = detail::async_rwlock_mutx_shared_state<value_type>
```

```cpp
template<>
using shared_state_ptr_type = std::shared_ptr<shared_state_type>
```

**Private Members**

```cpp
value_type value
allocator_type alloc
detail::async_rwlock_mutx_access_type prev_access = detail::async_rwlock_mutx_access_type::readwrite
shared_state_ptr_type prev_state
shared_state_ptr_type state
```

```cpp
template<detail::async_rwlock_mutx_access_type AccessType>
struct sender
```

**Public Types**

```cpp
template<>
template<>
using access_type = detail::async_rwlock_mutx_access_wrapper<readwrite_type, read_type, AccessType>
```

```cpp
template<>
template<template<typename...> class Tuple, template<typename...> class Variant>
using value_types = Variant<Tuple<access_type>>
```

```cpp
template<>
template<template<typename...> class Variant>
using error_types = Variant<std::exception_ptr>
```

**Public Members**

```cpp
template<>
shared_state_ptr_type prev_state
```

```cpp
template<>
shared_state_ptr_type state
```

**Public Static Attributes**

```cpp
template<>
constexpr bool sends_done = false
```
Friends

template<typename R>
auto tag_invoke(hpx::execution::experimental::connect_t, sender &&s, R &&r)

template<typename R>
struct operation_state

Public Functions

template<>
template<typename R_>
operation_state(R_ &&r, shared_state_ptr_type prev_state, shared_state_ptr_type state)

template<>
template<>
operation_state(operation_state&&)

template<>
template<>
operation_state(operation_state const&)

Public Members

template<>
template<>
std::decay_t<R> r

template<>
template<>
shared_state_ptr_type prev_state

template<>
template<>
shared_state_ptr_type state
Friends

```cpp
void tag_invoke(hpx::execution::experimental::start_t, operation_state &os)
```

template<typename Allocator>
class async_rw_mutex<void, void, Allocator>

Public Types

```cpp
template<> using read_type = void
template<> using readwrite_type = void
template<> using read_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::async_rw_mutex_access_type::read>
template<> using readwrite_access_type = detail::async_rw_mutex_access_wrapper<readwrite_type, read_type, detail::async_rw_mutex_access_type::readwrite>
template<> using allocator_type = Allocator
```

Public Functions

```cpp
async_rw_mutex<>(allocator_type const &alloc = {}) 
async_rw_mutex<async_rw_mutex&&>
async_rw_mutex& operator=(async_rw_mutex&&)
async_rw_mutex<async_rw_mutex const&>
async_rw_mutex& operator=(async_rw_mutex const&)
sender<detail::async_rwlock_access_type::read> read()
sender<detail::async_rwlock_access_type::readwrite> readwrite()
```

Private Types

```cpp
template<> using shared_state_type = detail::async_rwlock_shared_state<void>
template<> using shared_state_ptr_type = std::shared_ptr<shared_state_type>
```
Private Members

allocator_type alloc
detail::async_rwlock_mutex_access_type prev_access = detail::async_rwlock_mutex_access_type::readwrite
shared_state_ptr_type prev_state
shared_state_ptr_type state

template<detail::async_rwlock_mutex_access_type AccessType>
  struct sender

Public Types

template<>
template<>
  using access_type = detail::async_rwlock_mutex_access_wrapper<readwrite_type, read_type, AccessType>

template<>
template<template<typename...> class Tuple, template<typename...> class Variant>
  using value_types = Variant<Tuple<access_type>>

template<>
template<template<typename...> class Variant>
  using error_types = Variant<std::exception_ptr>

Public Members

template<>
  shared_state_ptr_type prev_state

template<>
  shared_state_ptr_type state

Public Static Attributes

template<>
  constexpr bool sends_done = false

Friends

template<typename R>
auto tag_invoke (hpx::execution::experimental::connect_t, sender &&s, R &&r)

template<typename R>
struct operation_state
Public Functions

```cpp
template<>
template<typename R_>
operation_state (R_ && r,
                 shared_state_ptr_type prev_state,
                 shared_state_ptr_type state)
```

```cpp
template<>
template<>
operation_state (operation_state&&)
```

```cpp
template<>
template<>
operation_state (operation_state const&)
```

```cpp
template<>
template<>
operation_state & operator= (operation_state&&)
```

```cpp
template<>
template<>
operation_state (operation_state const&)
```

Public Members

```cpp
template<>
template<>
std::decay_t<R> r
```

```cpp
template<>
template<>
shared_state_ptr_type prev_state
```

```cpp
template<>
template<>
shared_state_ptr_type state
```

Friends

```cpp
void tag_invoke (hpx::execution::experimental::start_t, operation_state & os)
```

namespace hpx

```cpp
namespace lcos
```

```cpp
namespace local
```

class barrier

```
#include <barrier.hpp> A barrier can be used to synchronize a specific number of threads, blocking all of the entering threads until all of the threads have entered the barrier.
```
**Note** A *barrier* is not a LCO in the sense that it has no global id and it can’t be triggered using the action (parcel) mechanism. It is just a low level synchronization primitive allowing to synchronize a given number of *threads*.

### Public Functions

**barrier** *(std::size_t number_of_threads)*

~**barrier** ()

void **wait** ()

  The function *wait* will block the number of entering *threads* (as given by the constructor parameter *number_of_threads*), releasing all waiting threads as soon as the last *thread* entered this function.

void **count_up** ()

  The function *count_up* will increase the number of *threads* to be waited in *wait* function.

void **reset** *(std::size_t number_of_threads)*

  The function *reset* will reset the number of *threads* as given by the function parameter *number_of_threads*. the newer coming *threads* executing the function *wait* will be waiting until *total_* is equal to *barrier_flag*. The last *thread* exiting the *wait* function will notify the newer *threads* waiting and the newer *threads* will get the reset *number_of_threads_* The function *reset* can be executed while previous *threads* executing waiting after they have been waken up. Thus *total_* can not be reset to *barrier_flag* which will break the comparison condition under the function *wait*.

### Private Types

**typedef** lcos::local::spinlock **mutex_type**

### Private Members

std::size_t **number_of_threads**

std::size_t **total**

**mutex_type** **mtx**

local::detail::condition_variable **cond**

### Private Static Attributes

**constexpr** std::size_t **barrier_flag** = static_cast<std::size_t>(1) << (CHAR_BIT * sizeof(std::size_t) - 1)

template<ctypename OnCompletion = detail::empty_oncompletion>

**class** **cpp20_barrier**
Public Types

template<>
using arrival_token = bool

Public Functions

HPX_NON_COPYABLE (cpp20_barrier)
cpp20_barrier (std::ptrdiff_t expected, OnCompletion completion = OnCompletion())
HPX_NODISCARD arrival_token hpx::lcos::local::cpp20_barrier::arrive(std::ptrdiff_t)

void wait (arrival_token &&old_phase) const

void arrive_and_wait ()
    Effects: Equivalent to: wait(arrive()).

void arrive_and_drop ()

Public Static Functions

static constexpr std::ptrdiff_t() hpx::lcos::local::cpp20_barrier::max()

Private Types

template<>
using mutex_type = lcos::local::spinlock

Private Functions

HPX_NODISCARD arrival_token hpx::lcos::local::cpp20_barrier::arrive_locked(std::unique_lock< mutex_type > & l, std::ptrdiff_t update = 1)

Private Members

mutex_type mtx_
local::detail::condition_variable cond_
std::ptrdiff_t expected_
std::ptrdiff_t arrived_
OnCompletion completion_
bool phase_

namespace hpx

namespace lcos

namespace local

Chapter 2. What’s so special about HPX?
Typedefs

template<typename T>
using channel_mPMC = bounded_channel<T, hpx::lcos::local::spinlock>

template<typename T, typename Mutex = util::spinlock>
class bounded_channel

Public Functions

bounded_channel (std::size_t size)
bounded_channel (bounded_channel &&rhs)
bounded_channel &operator= (bounded_channel &&rhs)
~bounded_channel ()
bool get (T *val = nullptr) const
bool set (T &t)
std::size_t close ()
std::size_t capacity () const

Protected Functions

std::size_t close (std::unique_lock<mutex_type> &l)

Private Types

template<>
using mutex_type = Mutex

Private Functions

bool is_full (std::size_t tail) const
bool is_empty (std::size_t head) const

Private Members

hpx::util::cache_aligned_data<mutex_type> mtx_
hpx::util::cache_aligned_data<std::size_t> head_
hpx::util::cache_aligned_data<std::size_t> tail_
std::size_t size_
std::unique_ptr<T[]> buffer_
bool closed_
namespace hpx

namespace lcos

namespace local

**Typedefs**

template<typename T>
using channel_mpsc = base_channel_mpsc<T, hpx::lcos::local::spinlock>

template<typename T, typename Mutex = util::spinlock>
class base_channel_mpsc

**Public Functions**

base_channel_mpsc(std::size_t size)

base_channel_mpsc(base_channel_mpsc &&rhs)

base_channel_mpsc &operator=(base_channel_mpsc &&rhs)

~base_channel_mpsc()

bool get(T *val = nullptr) const

bool set(T &t)

std::size_t close()

std::size_t capacity() const

**Private Types**

template<>
using mutex_type = Mutex

**Private Functions**

bool is_full(std::size_t tail) const

bool is_empty(std::size_t head) const
Private Members

```
hpx::util::cache_aligned_data<std::atomic<std::size_t>> head_
hpx::util::cache_aligned_data<tail_data> tail_
std::size_t size_
std::unique_ptr<T[]> buffer_
std::atomic<bool> closed_
```

```
struct tail_data
```

Public Members

```
template<> mutex_type mtx_
template<> std::atomic<std::size_t> tail_
```

```
namespace hpx
```

```
namespace lcos
```

```
namespace local
```

```
template<typename T>
class channel_spsc
```

Public Functions

```
channel_spsc(std::size_t size)
channel_spsc(channel_spsc &&rhs)
channel_spsc &operator=(channel_spsc &&rhs)
~channel_spsc()
bool get (T *val = nullptr) const
bool set (T &t)
std::size_t close ()
std::size_t capacity () const
```

2.8. API reference 1173
Private Functions

bool is_full (std::size_t tail) const
bool is_empty (std::size_t head) const

Private Members

hpx::util::cache_aligned_data<std::atomic<std::size_t>> head_

hpx::util::cache_aligned_data<std::atomic<std::size_t>> tail_

std::size_t size_

std::unique_ptr<T[]> buffer_

std::atomic<bool> closed_

namespace hpx

namespace lcos

namespace local

Enums

enum cv_status

Values:

no_timeout
timeout
error

class condition_variable

Public Functions

condition_variable() const
~condition_variable() const

void notify_one (error_code &ec = throws)

void notify_all (error_code &ec = throws)

template<typename Mutex>
void wait (std::unique_lock<Mutex> &lock, error_code &ec = throws)

template<typename Mutex, typename Predicate>
void wait (std::unique_lock<Mutex> &lock, Predicate pred, error_code &ec = throws)

template<typename Mutex>

cv_status wait_until (std::unique_lock<Mutex> &lock, hpx::chrono::steady_time_point const &abs_time, error_code &ec = throws)
template<typename Mutex, typename Predicate>
bool wait_until (std::unique_lock<Mutex> &lock, hpx::chrono::steady_time_point<const &abs_time, Predicate pred, error_code &ec = throws)

template<typename Mutex>
cv_status wait_for (std::unique_lock<Mutex> &lock, hpx::chrono::steady_duration const &rel_time, error_code &ec = throws)

Private Types

using mutex_type = detail::condition_variable_data::mutex_type
using data_type = hpx::memory::intrusive_ptr<detail::condition_variable_data>

Private Members

hpx::util::cache_aligned_data_derived<data_type> data_

class condition_variable_any

Public Functions

condition_variable_any ()
~condition_variable_any ()

void notify_one (error_code &ec = throws)
void notify_all (error_code &ec = throws)

template<typename Lock>
void wait (Lock &lock, error_code &ec = throws)

template<typename Lock, typename Predicate>
void wait (Lock &lock, Predicate pred, error_code & = throws)

template<typename Lock>
cv_status wait_until (Lock &lock, hpx::chrono::steady_time_point const &abs_time, error_code &ec = throws)

template<typename Lock, typename Predicate>
bool wait_until (Lock &lock, hpx::chrono::steady_time_point const &abs_time, Predicate pred, error_code &ec = throws)

template<typename Lock>
cv_status wait_for (Lock &lock, hpx::chrono::steady_duration const &rel_time, error_code &ec = throws)

template<typename Lock, typename Predicate>
bool wait_for (Lock &lock, hpx::chrono::steady_duration const &rel_time, Predicate pred, error_code &ec = throws)

template<typename Lock, typename Predicate>
bool wait (Lock &lock, stop_token stoken, Predicate pred, error_code &ec = throws)

template<typename Lock, typename Predicate>
bool wait_until (Lock &lock, stop_token stoken, hpx::chrono::steady_time_point const &abs_time, Predicate pred, error_code &ec = throws)

template<typename Lock, typename Predicate>
bool wait_for (Lock &lock, stop_token stoken, hpx::chrono::steady_duration const &rel_time, Predicate pred, error_code &ec = throws)

Private Types

using mutex_type = detail::condition_variable_data::mutex_type
using data_type = hpx::memory::intrusive_ptr<detail::condition_variable_data>

Private Members

hpx::util::cache_aligned_data_derived<data_type> data_

namespace hpx

namespace lcos

namespace local

Typedefs

typedef counting_semaphore_var counting_semaphore

template<typename Mutex = hpx::lcos::local::spinlock, int N = 0>
class counting_semaphore_var : private hpx::lcos::local::cpp20_counting_semaphore<PTRDIFF_MAX, hpx::lcos::local::spinlock, N, std::mutex_traits<Mutex>>

Public Functions

counting_semaphore_var (std::ptrdiff_t value = N)
counting_semaphore_var (counting_semaphore_var const&)
counting_semaphore_var &operator= (counting_semaphore_var const&)
void wait (std::ptrdiff_t count = 1)
bool try_wait (std::ptrdiff_t count = 1)
void signal (std::ptrdiff_t count = 1)
  Signal the semaphore.
std::ptrdiff_t signal_all ()

Chapter 2. What’s so special about HPX?
Private Types

```cpp
template<>
using mutex_type = Mutex

template<typename Mutex = hpx::lcos::local::spinlock>
class cpp20_binary_semaphore : public hpx::lcos::local::cpp20_counting_semaphore<1, hpx::lcos::local::spinlock>
```

Public Functions

```cpp
HPX_NON_COPYABLE (cpp20_binary_semaphore)
cpp20_binary_semaphore (std::ptrdiff_t value = 1)
~cpp20_binary_semaphore ()
```

```cpp
template<std::ptrdiff_t LeastMaxValue = PTRDIFF_MAX, typename Mutex = hpx::lcos::local::spinlock>
class cpp20_counting_semaphore
```

Public Functions

```cpp
HPX_NON_COPYABLE (cpp20_counting_semaphore)
cpp20_counting_semaphore (std::ptrdiff_t value)
~cpp20_counting_semaphore ()
void release (std::ptrdiff_t update = 1)
bool try_acquire ()
void acquire ()

bool try_acquire_until (hpx::chrono::steady_time_point const &abs_time)
bool try_acquire_for (hpx::chrono::steady_duration const &rel_time)
```

Public Static Functions

```cpp
static constexpr std::ptrdiff_t() hpx::lcos::local::cpp20_counting_semaphore::max()
```

Protected Types

```cpp
template<>
using mutex_type = Mutex
```
**Protected Attributes**

mutex_type mtx_

detail::counting_semaphore sem_

```cpp
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.

```cpp
// Public Functions
```

```cpp
event ()
   Construct a new event semaphore.

bool occurred ()
   Check if the event has occurred.

void wait ()
   Wait for the event to occur.

void set ()
   Release all threads waiting on this semaphore.

void reset ()
   Reset the event.

// Private Types
```

```cpp
typedef lcos::local::spinlock mutex_type
```

```cpp
// Private Functions
```

```cpp
void wait_locked (std::unique_lock<mutex_type> &l)

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_

namespace hpx

namespace lcos

namespace local

class cpp20_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

Public Functions

HPX_NON_COPYABLE (cpp20_latch)

cpp20_latch (std::ptrdiff_t count)
  Initialize the latch

  Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

~cpp20_latch ()

  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.

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.

Exceptions

• Nothing.

bool try_wait () const

  Returns: With very low probability false. Otherwise counter == 0.
void wait() const
    If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization point until counter_ reaches 0.

Exceptions
    • Nothing.

void arrive_and_wait(std::ptrdiff_t update = 1)
    Effects: Equivalent to: count_down(update); wait();

Public Static Functions

static constexpr std::ptrdiff_t() hpx::lcos::local::cpp20_latch::max()
    Returns: The maximum value of counter that the implementation supports.

Protected Types

using mutex_type = lcos::local::spinlock

Protected Attributes

util::cache_line_data<mutex_type> mtx_
util::cache_line_data<local::detail::condition_variable> cond_
std::atomic<std::ptrdiff_t> counter_
bool notified_

class latch: public hpx::lcos::local::cpp20_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 local::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 lcos::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)

latch(std::ptrdiff_t count)
    Initialize the latch
    Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

~latch()
    Requires: No threads are blocked at the synchronization point.
May be called even if some threads have not yet returned from `wait()` or
`count_down_and_wait()`, provided that `counter_` is 0.

The destructor might not return until all threads have exited `wait()` or
`count_down_and_wait()`.

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.

```cpp
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.

Exceptions
    • Nothing.

bool is_ready() const
    Returns: `counter_` == 0. Does not block.

Exceptions
    • Nothing.

void abort_all()

void count_up(std::ptrdiff_t n)
    Increments `counter_` by n. Does not block.
    Requires: n >= 0.

Exceptions
    • Nothing.

void reset(std::ptrdiff_t n)
    Reset `counter_` to n. Does not block.
    Requires: n >= 0.

Exceptions
    • Nothing.
```

namespace hpx

namespace lcos

namespace local
Functions

template<typename Mutex>
void swap(upgrade_lock<Mutex> &lhs, upgrade_lock<Mutex> &rhs)

template<typename Mutex>
class upgrade_lock

Public Types

template<>
using mutex_type = Mutex

Public Functions

upgrade_lock (upgrade_lock const&)
upgrade_lock &operator= (upgrade_lock const&)
upgrade_lock ()
upgrade_lock (Mutex &m_)
upgrade_lock (Mutex &m_, std::adopt_lock_t)
upgrade_lock (Mutex &m_, std::defer_lock_t)
upgrade_lock (Mutex &m_, std::try_to_lock_t)
upgrade_lock (upgrade_lock<Mutex> &&other)
upgrade_lock (std::unique_lock<Mutex> &&other)
upgrade_lock &operator= (upgrade_lock<Mutex> &&other)
void swap (upgrade_lock &other)
Mutex *mutex () const
Mutex *release ()
~upgrade_lock ()
void lock ()
bool try_lock ()
void unlock ()
operator bool () const
bool owns_lock () const
**Protected Attributes**

Mutex *m
bool is_locked

**Friends**

friend hpx::lcos::local::upgrade_to_unique_lock

template<typename Mutex>
class upgrade_to_unique_lock

**Public Types**

template<>
using mutex_type = Mutex

**Public Functions**

upgrade_to_unique_lock (upgrade_to_unique_lock const&)
upgrade_to_unique_lock &operator= (upgrade_to_unique_lock const&)
upgrade_to_unique_lock (upgrade_lock<Mutex> &m_)
~upgrade_to_unique_lock()
upgrade_to_unique_lock (upgrade_to_unique_lock<Mutex> &&other)
upgrade_to_unique_lock &operator= (upgrade_to_unique_lock<Mutex> &&other)
void swap (upgrade_to_unique_lock &other)
operator bool () const
bool owns_lock () const
Mutex *mutex () const

**Private Members**

upgrade_lock<Mutex> *source
std::unique_lock<Mutex> exclusive

namespace hpx

namespace lcos

namespace local

class mutex
Subclassed by hpx::lcos::local::timed_mutex
Public Functions

HPX_NON_COPYABLE (mutex)

mutex (char const *const description = "")

~mutex ()

void lock (char const *description, error_code &ec = throws)
void lock (error_code &ec = throws)
bool try_lock (char const *description, error_code &ec = throws)
bool try_lock (error_code &ec = throws)
void unlock (error_code &ec = throws)

Protected Types

typedef lcos::local::spinlock mutex_type

Protected Attributes

mutex_type mtx_
threads::thread_id_type owner_id_
lcos::local::detail::condition_variable cond_

class timed_mutex : private hpx::lcos::local::mutex

Public Functions

HPX_NON_COPYABLE (timed_mutex)

timed_mutex (char const *const description = "")

~timed_mutex ()

bool try_lock_until (hpx::chrono::steady_time_point const &abs_time, char const *description, error_code &ec = throws)
bool try_lock_until (hpx::chrono::steady_time_point const &abs_time, error_code &ec = throws)
bool try_lock_for (hpx::chrono::steady_duration const &rel_time, char const *description, error_code &ec = throws)
bool try_lock_for (hpx::chrono::steady_duration const &rel_time, error_code &ec = throws)
void lock (char const *description, error_code &ec = throws)
void lock (error_code &ec = throws)
bool try_lock (char const *description, error_code &ec = throws)
bool **try_lock**(*error_code &ec = throws*)

```c
void unlock(*error_code &ec = throws*)
```

### Typedefs

```c
using thread_id_ref_type = thread_id_ref
using thread_self = coroutines::detail::coroutine_self
```

### Functions

```c
thread_id get_self_id()
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).
```

```c
thread_self *get_self_ptr()
The function get_self_ptr returns a pointer to the (OS thread specific) self reference to the current HPX thread.
```

### Structures

```c
namespace hpx
```

```c
namespace lcos
```

```c
namespace local
```

```c
struct no_mutex
```

### Public Functions

```c
void lock()
bool try_lock()
void unlock()
```

### Defines

```c
HPX_ONCE_INIT
```

```c
namespace hpx
```

```c
namespace lcos
```

```c
namespace local
```
Functions

template< typename F, typename ...Args >
void call_once ( once_flag & flag, F && f, Args &&... args )

struct once_flag

Public Functions

HPX_NON_COPYABLE ( once_flag )

once_flag ()

Private Members

std::atomic<long> status_

lcos::local::event event_

Friends

template< typename F, typename ...Args >
void call_once ( once_flag & flag, F && f, Args &&... args )

namespace hpx

namespace lcos

namespace local

Typedefs

using recursive_mutex = detail::recursive_mutex_impl<>

namespace hpx

namespace lcos

namespace local
Typedef

typedef detail::shared_mutex shared_mutex

namespace hpx

namespace lcos

namespace local

Typedef

typedef sliding_semaphore_var sliding_semaphore

template<typename Mutex = hpx::lcos::local::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 (std::int64_t max_difference, std::int64_t lower_limit = 0) 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.

void set_max_difference (std::int64_t max_difference, std::int64_t lower_limit = 0) 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()).

2.8. API reference
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.

bool try_wait (std::int64_t upper_limit = 1)
Try to wait for the semaphore to be signaled.

Return The function returns true if the calling thread would not block if it was calling wait().

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.

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.

std::int64_t signal_all ()

Private Types
typedef Mutex mutex_type

Private Members
mutex_type mtx_

detail::sliding_semaphore sem_

namespace hpx

namespace lcos

namespace local

struct spinlock
Public Functions

HPX_NON_COPYABLE (spinlock)

spinlock (char const *const desc = "hpx::lcos::local::spinlock")

~spinlock()
void lock()
bool try_lock()
void unlock()

Private Functions

bool acquire_lock()
void relinquish_lock()
bool is_locked() const

Private Members

std::atomic<bool> v_

namespace hpx

namespace lcos

namespace local

struct spinlock_no_backoff
#include <spinlock_no_backoff.hpp> boost::mutex-compatible spinlock class

Public Functions

HPX_NON_COPYABLE (spinlock_no_backoff)

spinlock_no_backoff()

~spinlock_no_backoff()
void lock()
bool try_lock()
void unlock()
Private Functions

bool acquire_lock()
void relinquish_lock()
bool is_locked() const

Private Members

namespace hpx

namespace lcos

namespace local

template<typename Tag, std::size_t N = HPX_HAVE_SPINLOCK_POOL_NUM>
class spinlock_pool

Public Static Functions

static lcos::local::spinlock &spinlock_for (void const *pv)

Private Static Attributes

util::cache_aligned_data<lcos::local::spinlock> pool_

class scoped_lock

Public Functions

template<>
HPX_NON_COPYABLE (scoped_lock)

template<>
scoped_lock (void const *pv)

template<>
~scoped_lock ()

template<>
void lock ()

template<>
void unlock ()
Private Members

template<>
    hpx::lcos::local::spinlock & sp_

namespace hpx

Functions

template<typename Callback>
    stop_callback<
typename std::decay<Callback>::type> make_stop_callback (stop_token const &st, Callback &cb)

template<typename Callback>
    stop_callback<
typename std::decay<Callback>::type> make_stop_callback (stop_token &&st, Callback &&cb)

    void swap (stop_token &lhs, stop_token &rhs)

    void swap (stop_source &lhs, stop_source &rhs)

Variables

HPX_INLINE_CONSTEXPR_VARIABLE nostopstate_t hpx::nostopstate = {}

struct nostopstate_t

Public Functions

    nostopstate_t ()

template<typename Callback>
    class stop_callback : private hpx::detail::stop_callback_base

Public Types

    template<>
        using callback_type = Callback

Public Functions

    template<typename CB, typename Enable = typename std::enable_if<std::is_constructible<Callback, CB>::value>::type>
        stop_callback (stop_token const &st, CB &&cb)

    template<typename CB, typename Enable = typename std::enable_if<std::is_constructible<Callback, CB>::value>::type>
        stop_callback (stop_token &&st, CB &&cb)

    ~stop_callback ()
stop_callback (stop_callback const&)
stop_callback (stop_callback&&)
stop_callback &operator= (stop_callback const&)
stop_callback &operator= (stop_callback&&)

Private Functions

void execute ()

Private Members

Callback callback_

hpx::memory::intrusive_ptr<detail::stop_state> state_

class stop_source

Public Functions

stop_source ()
stop_source (nostopstate_t)
stop_source (stop_source const &rhs)
stop_source (stop_source&&)
stop_source &operator= (stop_source const &rhs)
stop_source &operator= (stop_source&&)
~stop_source ()
void swap (stop_source &s)
HPX_NODISCARD stop_token hpx::stop_source::get_token() const
HPX_NODISCARD bool hpx::stop_source::stop_possible() const
HPX_NODISCARD bool hpx::stop_source::stop_requested() const
bool request_stop ()

Private Members

hpx::memory::intrusive_ptr<detail::stop_state> state_
class stop_token

Public Functions

stop_token()
stop_token(stop_token const & rhs)
stop_token(stop_token&&)
stop_token &operator=(stop_token const & rhs)
stop_token &operator=(stop_token&&)
~stop_token()
void swap(stop_token &s)
HPX_NODISCARD bool hpx::stop_token::stop_requested() const
HPX_NODISCARD bool hpx::stop_token::stop_possible() const

Private Functions

stop_token(hpx::memory::intrusive_ptr<detail::stop_state> const & state)

Private Members

hpx::memory::intrusive_ptr<detail::stop_state> state_

Friends

friend hpx::stop_callback
friend hpx::stop_source
HPX_NODISCARD friend bool operator==(stop_token const & lhs, stop_token const & rhs)
HPX_NODISCARD friend bool operator!=(stop_token const & lhs, stop_token const & rhs)
tag_invoke

The contents of this module can be included with the header `hpx/modules/tag_invoke.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/tag_invoke.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace util

Typedefs

template<typename F, typename ...Ts>
using invoke_result_t = typename invoke_result<F, Ts...>::type
```

```cpp
namespace hpx

namespace functional

Typedefs

template<typename Tag, typename ...Args>
using tag_invoke_result = invoke_result<decltype(tag_invoke), Tag, Args...>

hpx::functional::tag_invoke_result<Tag, Args...> is the trait returning the result type of the call hpx::functional::tag_invoke. This can be used in a SFINAE context.

template<typename Tag, typename ...Args>
using tag_invoke_result_t = typename tag_invoke_result<Tag, Args...>::type

hpx::functional::tag_invoke_result_t<Tag, Args...>::type evaluates to hpx::functional::tag_invoke_result<Tag, Args...>::type
```

Variables

```cpp
constexpr unspecified tag_invoke = unspecified
```

The hpx::functional::tag_invoke name defines a constexpr object that is invocable with one or more arguments. The first argument is a ‘tag’ (typically a DPO). It is only invocable if an overload of tag_invoke() that accepts the same arguments could be found via ADL.

The evaluation of the expression hpx::tag_invoke(tag, args...) is equivalent to evaluating the unqualified call to tag_invoke(decay-copy(tag), std::forward<Args>(args)...).

hpx::functional::tag_invoke is implemented against P1895.

Example: Defining a new customization point foo:

```cpp
namespace mylib {
    inline constexpr
        struct foo_fn final : hpx::functional::tag<foo_fn>
        {
            } foo{};
}
```
Defining an object \texttt{bar} which customizes \texttt{foo}:

```cpp
struct bar
{
    int \_x = 42;

    friend constexpr int tag_invoke(mylib::foo\_fn, bar \_const & \_x)
    {
        return \_b.\_x;
    }
};
```

Using the customization point:

```cpp
static_assert(42 == mylib::foo(bar{}), "The answer is 42");
```

\texttt{template<typename Tag, typename ...Args> constexpr bool is\_tag\_invocable\_v = is\_tag\_invocable<Tag, Args...>::value}
\texttt{hpx::functional::is\_tag\_invocable\_v<Tag, Args...> evaluates to}
\texttt{hpx::functional::is\_tag\_invocable<Tag, Args...>::value}

\texttt{template<typename Tag, typename ...Args> constexpr bool is\_nothrow\_tag\_invocable\_v = is\_nothrow\_tag\_invocable<Tag, Args...>::value}
\texttt{hpx::functional::is\_tag\_invocable\_v<Tag, Args...> evaluates to}
\texttt{hpx::functional::is\_tag\_invocable<Tag, Args...>::value}

\texttt{template<typename Tag, typename ...Args> struct is\_nothrow\_tag\_invocable}
\texttt{#include <tag\_invoke.hpp> hpx::functional::is\_nothrow\_tag\_invocable\_v<Tag, Args...> is std::true\_type if an overload of tag\_invoke(tag, args...) can be found via ADL and is noexcept.}

\texttt{template<typename Tag, typename ...Args> struct is\_tag\_invocable}
\texttt{#include <tag\_invoke.hpp> hpx::functional::is\_tag\_invocable\_v<Tag, Args...> is std::true\_type if an overload of tag\_invoke(tag, args...) can be found via ADL.}

\texttt{template<typename Tag> struct tag}
\texttt{#include <tag\_invoke.hpp> hpx::functional::tag\_Tag defines a base class that implements the necessary tag dispatching functionality for a given type Tag}

\texttt{template<typename Tag> struct tag\_noexcept}
\texttt{#include <tag\_invoke.hpp> hpx::functional::tag\_noexcept\_Tag defines a base class that implements the necessary tag dispatching functionality for a given type Tag The implementation has to be noexcept}

\texttt{namespace hpx}
Variables

```cpp
template<typename F, typename... Ts> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::is_invocable_v = is_invocable<F, Ts...>::value
```

```cpp
template<typename R, typename F, typename... Ts> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::is_invocable_r_v = is_invocable_r<R, F, Ts...>::value
```

namespace traits

**Typedefs**

```cpp
typedef hpx::is_invocable_r<R, F, Ts...> instead
```

**testing**

The contents of this module can be included with the header `hpx/modules/testing.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/testing.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

**Defines**

```cpp
HPX_TEST(...)  
HPX_TEST_(...)  
HPX_TEST_1(expr)  
HPX_TEST_2(stream, expr)  
HPX_TEST_IMPL(fixture, expr)  
HPX_TEST_MSG(...)  
HPX_TEST_MSG_(...)  
HPX_TEST_MSG_2(expr, msg)  
HPX_TEST_MSG_3(stream, expr, msg)  
HPX_TEST_MSG_IMPL(fixture, expr, msg)  
HPX_TEST_EQ(...)  
HPX_TEST_EQ_(...)  
HPX_TEST_EQ_2(expr1, expr2)  
HPX_TEST_EQ_3(stream, expr1, expr2)  
HPX_TEST_EQ_IMPL(fixture, expr1, expr2)  
HPX_TEST_NEQ(...)  
HPX_TEST_NEQ_(...)  
HPX_TEST_NEQ_2(expr1, expr2)  
HPX_TEST_NEQ_3(stream, expr1, expr2)  
HPX_TEST_NEQ_IMPL(fixture, expr1, expr2)  
HPX_TEST_LT(...)  
```

Chapter 2. What’s so special about HPX?
HPX_TEST_LT_(...) 
HPX_TEST_LT_2(expr1, expr2) 
HPX_TEST_LT_3(strm, expr1, expr2) 
HPX_TEST_LT_IMPL(fixture, expr1, expr2) 
HPX_TEST_LTE(...) 
HPX_TEST_LTE_(...) 
HPX_TEST_LTE_2(expr1, expr2) 
HPX_TEST_LTE_3(strm, expr1, expr2) 
HPX_TEST_LTE_IMPL(fixture, expr1, expr2) 
HPX_TEST_RANGE(...) 
HPX_TEST_RANGE_(...) 
HPX_TEST_RANGE_3(expr1, expr2, expr3) 
HPX_TEST_RANGE_4(strm, expr1, expr2, expr3) 
HPX_TEST_RANGE_IMPL(fixture, expr1, expr2, expr3) 
HPX_TEST_EQ_MSG(...) 
HPX_TEST_EQ_MSG_(...) 
HPX_TEST_EQ_MSG_3(expr1, expr2, msg) 
HPX_TEST_EQ_MSG_4(strm, expr1, expr2, msg) 
HPX_TEST_EQ_MSG_IMPL(fixture, expr1, expr2, msg) 
HPX_TEST_NEQ_MSG(...) 
HPX_TEST_NEQ_MSG_(...) 
HPX_TEST_NEQ_MSG_3(expr1, expr2, msg) 
HPX_TEST_NEQ_MSG_4(strm, expr1, expr2, msg) 
HPX_TEST_NEQ_MSG_IMPL(fixture, expr1, expr2, msg) 
HPX_TEST_LT_MSG(...) 
HPX_TEST_LT_MSG_(...) 
HPX_TEST_LT_MSG_3(expr1, expr2, msg) 
HPX_TEST_LT_MSG_4(strm, expr1, expr2, msg) 
HPX_TEST_LT_MSG_IMPL(fixture, expr1, expr2, msg) 
HPX_TEST_LTE_MSG(...) 
HPX_TEST_LTE_MSG(...) 
HPX_TEST_LTE_MSG_3(expr1, expr2, msg) 
HPX_TEST_LTE_MSG_4(strm, expr1, expr2, msg) 
HPX_TEST_LTE_MSG_IMPL(fixture, expr1, expr2, msg) 
HPX_TEST_RANGE_MSG(...) 
HPX_TEST_RANGE_MSG_...
HPX_TEST_RANGE_MSG_4 (expr1, expr2, expr3, msg)
HPX_TEST_RANGE_MSG_5 (strm, expr1, expr2, expr3, msg)
HPX_TEST_RANGE_MSG_IMPL (fixture, expr1, expr2, expr3, msg)
HPX_SANITY (...) 
HPX_SANITY__ (...) 
HPX_SANITY_1 (expr) 
HPX_SANITY_2 (strm, expr) 
HPX_SANITY_IMPL (fixture, expr) 
HPX_SANITY_MSG (...) 
HPX_SANITY_MSG__ (...) 
HPX_SANITY_MSG_2 (expr, msg) 
HPX_SANITY_MSG_3 (strm, expr, msg) 
HPX_SANITY_MSG_IMPL (fixture, expr, msg) 
HPX_SANITY_EQ (...) 
HPX_SANITY_EQ__ (...) 
HPX_SANITY_EQ_2 (expr1, expr2) 
HPX_SANITY_EQ_3 (strm, expr1, expr2) 
HPX_SANITY_EQ_IMPL (fixture, expr1, expr2) 
HPX_SANITY_NEQ (...) 
HPX_SANITY_NEQ__ (...) 
HPX_SANITY_NEQ_2 (expr1, expr2) 
HPX_SANITY_NEQ_3 (strm, expr1, expr2) 
HPX_SANITY_NEQ_IMPL (fixture, expr1, expr2) 
HPX_SANITY_LT (...) 
HPX_SANITY_LT__ (...) 
HPX_SANITY_LT_2 (expr1, expr2) 
HPX_SANITY_LT_3 (strm, expr1, expr2) 
HPX_SANITY_LT_IMPL (fixture, expr1, expr2) 
HPX_SANITY_LTE (...) 
HPX_SANITY_LTE__ (...) 
HPX_SANITY_LTE_2 (expr1, expr2) 
HPX_SANITY_LTE_3 (strm, expr1, expr2) 
HPX_SANITY_LTE_IMPL (fixture, expr1, expr2) 
HPX_SANITY_RANGE (...) 
HPX_SANITY_RANGE__ (...) 
HPX_SANITY_RANGE_3 (expr1, expr2, expr3)
namespace hpx

namespace util

**Typedefs**

```cpp
using test_failure_handler_type = function_nonser<void>()
```

** Enums**

```cpp
class counter_type
{
    Values:
    counter_sanity
    counter_test
}
```

** Functions**

```cpp
void set_test_failure_handler(test_failure_handler_type f)
int report_errors()
int report_errors(std::ostream &stream)
void print_cdash_timing(const char *name, double time)
void print_cdash_timing(const char *name, std::uint64_t time)
```

namespace hpx

namespace util
Functions

```cpp
void perftests_report(std::string const &name, std::string const &exec, const std::size_t steps, function_nonser<void> &&test)
```

```cpp
void perftests_print_times()
```

### thread_pool_util

The contents of this module can be included with the header `hpx/modules/thread_pool_util.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/thread_pool_util.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
```

```cpp
namespace threads
```

### Functions

```cpp
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.

**Return** A `future<void>` which is ready when the given 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.

```cpp
void resume_processing_unit_cb(thread_pool_base &pool, util::function_nonser<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**

- `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.

```cpp
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.

**Return** A `future<void>` which is ready when the given processing unit has been suspended.

**Parameters**
- `virt_core`: [in] The processing unit on the pool to be suspended. The processing units are indexed starting from 0.

**Exceptions**
- `hpx::exception`: if called from outside the HPX runtime.

```cpp
void suspend_processing_unit_cb (util::function_nonser<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**
- `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.

```cpp
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.

**Return** A `future<void>` which is ready when the thread pool has been resumed.

**Exceptions**
- `hpx::exception`: if called from outside the HPX runtime.

```cpp
void resume_pool_cb (thread_pool_base &pool, util::function_nonser<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**
- `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.

```cpp
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.

**Return** A `future<void>` which is ready when the thread pool has been suspended.

**Exceptions**
- `hpx::exception`: if called from outside the HPX runtime.
void suspend_pool_cb (thread_pool_base &pool, util::function_nonser<void>)
> callback, error_code &ec = throwsSuspends 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
• 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.

Exceptions
• hpx::exception: if called from an HPX thread which is running on the pool itself.

thread_pools

The contents of this module can be included with the header hpx/modules/thread_pools.hpp. These headers
may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these
at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including
the module header hpx/modules/thread_pools.hpp, not the particular header in which the functionality you
would like to use is defined. See Public API for a list of names that are part of the public HPX API.

thread_support

The contents of this module can be included with the header hpx/modules/thread_support.hpp. These headers
may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these
at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including
the module header hpx/modules/thread_support.hpp, not the particular header in which the functionality you
would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

HPX_ASSERT_OWNS_LOCK (l)
HPX_ASSERT_DOESNT_OWN_LOCK (l)

namespace hpx

namespace util

class atomic_count

Public Functions

HPX_NON_COPYABLE (atomic_count)
atomic_count (long value)
atomic_count &operator= (long value)
long operator++ ()
long operator-- ()
atomic_count &operator+=(long n)
atomic_count &operator-=(long n)
operator long() const

Private Members

std::atomic<long> value_

namespace hpx

namespace util

Functions

void set_thread_name(char const*)

Defines

HPX_CORE_EXPORT_THREAD_SPECIFIC_PTR

namespace hpx

namespace util

template<typename T, typename Tag>
struct thread_specific_ptr

Public Types

typedef T element_type

Public Functions

T *get() const
T *operator->() const
T &operator*() const
void reset(T *new_value = nullptr)
Private Static Attributes

thread_local T *ptr_ = nullptr

namespace hpx

namespace util

template<typename Mutex>
class unlock_guard

Public Types

template<>
using mutex_type = Mutex

Public Functions

HPX_NON_COPYABLE (unlock_guard)

unlock_guard (Mutex &m)

~unlock_guard()

Private Members

Mutex &m_

threading

The contents of this module can be included with the header hpx/modules/threading.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/threading.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

Functions

void swap (jthread &lhs, jthread &rhs)

class jthread
Public Types

using id = thread::id
using native_handle_type = thread::native_handle_type

Public Functions

jthread()

template<
typename F, typename ...Ts
typename Enable = typename std::enable_if<!std::is_same<
typename std::decay<F>::type,
jthread>::value>::type>

jthread(
F &&f, Ts&&... ts)

~jthread()

jthread(jthread const&)  
jthread(jthread &&x)

jthread &operator=(jthread const&)  
jthread &operator=(jthread&&)

void swap(jthread &t)

HPX_NODISCARD bool hpx::jthread::joinable() const

void join()

void detach()

HPX_NODISCARD id hpx::jthread::get_id() const

HPX_NODISCARD native_handle_type hpx::jthread::native_handle()

HPX_NODISCARD stop_source hpx::jthread::get_stop_source()

HPX_NODISCARD stop_token hpx::jthread::get_stop_token() const

bool request_stop()

Public Static Functions

static HPX_NODISCARD unsigned int hpx::jthread::hardware_concurrency()

Private Members

stop_source ssoure_  
hpx::thread thread_ = []
Private Static Functions

template<typename F, typename ...Ts>
static void invoke (std::false_type, F &&f, stop_token &&, Ts&&... ts)

template<typename F, typename ...Ts>
static void invoke (std::true_type, F &&f, stop_token &&st, Ts&&... ts)

struct hash<::hpx::thread::id>

Public Functions

std::size_t operator () (::hpx::thread::id const &id) const

namespace hpx

Typedefs

using thread_termination_handler_type = util::function_nonser<void (std::exception_ptr const &e) >

Functions

void set_thread_termination_handler (thread_termination_handler_type f)

void swap (thread &x, thread &y)

bool operator== (thread::id const &x, thread::id const &y)

bool operator!= (thread::id const &x, thread::id const &y)

bool operator< (thread::id const &x, thread::id const &y)

bool operator> (thread::id const &x, thread::id const &y)

bool operator<= (thread::id const &x, thread::id const &y)

bool operator>= (thread::id const &x, thread::id const &y)

template<typename Char, typename Traits>
std::basic_ostream<Char, Traits> &operator<< (std::basic_ostream<Char, Traits> &out, thread::id const &id)

class thread
Public Types

typedef threads::thread_id_type native_handle_type

Public Functions

thread()

template<typename F, typedef typename std::enable_if<!std::is_same<typename decay<F>::type, threads::thread>::value>::type>
thread(F &&f)

template<typename F, typename ...Ts>
thread(F &&f, Ts&&... vs)

template<typename F>
thread(threads::thread_pool_base *pool, F &&f)

template<typename F, typename ...Ts>
thread(threads::thread_pool_base *pool, F &&f, Ts&&... vs)

~thread()

thread(thread&&)

thread &operator=(thread&&)

void swap(thread&)

bool joinable() const

void join()

void detach()

id get_id() const

native_handle_type native_handle() const

void interrupt(bool flag = true)

bool interruption_requested() const

lcos::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 HPX_NODISCARD unsigned int hpx::thread::hardware_concurrency()
Private Types

typedef lcos::local::spinlock mutex_type

Private Functions

void terminate(const char *function, const char *reason) const

bool joinable_locked() const

void detach_locked()

void start_thread(threads::thread_pool_base *pool, util::unique_function_nonser<void> > &func

Private Members

mutex_type mtx_

threads::thread_id_ref_type id_

Private Static Functions

static threads::thread_result_type thread_function_nullary (util::unique_function_nonser<void> > const &func

class id

Public Functions

id()

id(threads::thread_id_type const &i)

id(threads::thread_id_type &&i)

id(threads::thread_id_ref_type const &i)

id(threads::thread_id_ref_type &&i)

threads::thread_id_type const &native_handle() const

Private Members

threads::thread_id_type id_
# Friends

```cpp
friend hpx::thread

bool operator== (thread::id const &x, thread::id const &y)
bool operator!= (thread::id const &x, thread::id const &y)
bool operator< (thread::id const &x, thread::id const &y)
bool operator> (thread::id const &x, thread::id const &y)
bool operator<=(thread::id const &x, thread::id const &y)
bool operator>=(thread::id const &x, thread::id const &y)
```

```cpp
template<typename Char, typename Traits>
std::basic_ostream<Char, Traits> &operator<< (std::basic_ostream<Char, Traits> &out, thread::id const &id)
```

```cpp
namespace this_thread

# Functions

```cpp
thread::id get_id ()
void yield ()
void yield_to (thread::id)
threads::thread_priority get_priority ()
std::ptrdiff_t get_stack_size ()
void interruption_point ()
bool interruption_enabled ()
bool interruption_requested ()
void interrupt ()
void sleep_until (hpx::chrono::steady_time_point const &abs_time)
void sleep_for (hpx::chrono::steady_duration const &rel_time)
std::size_t get_thread_data ()
std::size_t set_thread_data (std::size_t)
```

```cpp
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 hpx::this_thread::restore_interruption

class restore_interruption

Public Functions

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<>
    struct hash<::hpx::thread::id>
Public Functions

```cpp
std::size_t operator() (::hpx::thread::id const &id) const
```

threading_base

The contents of this module can be included with the header `hpx/modules/threading_base.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/threading_base.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace util

Functions

```cpp
template<typename F>
constexpr F &&annotated_function (F &&f, char const* = nullptr)
```

Given a function as an argument, the user can `annotated_function` as well. Annotating includes setting the thread description per thread id.

Parameters

- `function`:

```cpp
template<typename F>
constexpr F &&annotated_function (F &&f, std::string const&)
```

struct annotate_function

Public Functions

```cpp
HPX_NON_COPYABLE (annotate_function)
```

```cpp
constexpr annotate_function (char const*)
```

```cpp
template<typename F>
constexpr annotate_function (F&&)
```

~annotate_function ()

```cpp
namespace hpx

namespace threads

namespace policies

class callback_notifier
```
Public Types

typedef util::function_nonser<void (std::size_t, std::size_t, char const*, char const*)> on_startstop_type

typedef util::function_nonser<bool (std::size_t, std::exception_ptr const&)> on_error_type

Public Functions

callback_notifier()

void on_start_thread (std::size_t local_thread_num, std::size_t global_thread_num, char const* pool_name, char const* postfix) const

void on_stop_thread (std::size_t local_thread_num, std::size_t global_thread_num, char const* pool_name, char const* postfix) const

bool on_error (std::size_t global_thread_num, std::exception_ptr const& e) const

void add_on_start_thread_callback (on_startstop_type const &callback)

void add_on_stop_thread_callback (on_startstop_type const &callback)

void set_on_error_callback (on_error_type const &callback)

Public Members

std::deque<on_startstop_type> on_start_thread_callbacks_

std::deque<on_startstop_type> on_stop_thread_callbacks_

on_error_type on_error_

namespace hpx

namespace threads

struct execution_agent : public agent_base

Public Functions

execution_agent (coroutines::detail::coroutine_impl *coroutine)

std::string description () const

execution_context const &context () const

void yield (char const *desc)

void yield_k (std::size_t k, char const *desc)

void suspend (char const *desc)

void resume (char const *desc)
void abort (char const *desc)
void sleep_for (hpx::chrono::steady_duration const &sleep_duration, char const *desc)
void sleep_until (hpx::chrono::steady_time_point const &sleep_time, char const *desc)

Private Functions

hpx::threads::thread_restart_state do_yield (char const *desc, threads::thread_schedule_state state)
void do_resume (char const *desc, hpx::threads::thread_restart_state statex)

Private Members

coroutines::detail::coroutine_stackful_self self_
execution_context context_

struct execution_context : public context_base

Public Functions

hpx::execution_base::resource_base const &resource () const

Public Members

hpx::execution_base::resource_base resource_

namespace hpx

namespace util

namespace external_timer

Functions

std::shared_ptr<task_wrapper> new_task (thread_description const &, std::uint32_t, threads::thread_id_type)

std::shared_ptr<task_wrapper> update_task (std::shared_ptr<task_wrapper>, thread_description const&)

struct scoped_timer
Public Functions

```cpp
scoped_timer (std::shared_ptr<task_wrapper>)

~scoped_timer ()

void stop (void)

void yield (void)
```

namespace hpx

```cpp
namespace threads
```

Functions

```cpp
template<typename F>
thread_function_type make_thread_function (F &&f)
```

```cpp
template<typename F>
thread_function_type make_thread_function_nullary (F &&f)
```

```cpp
threads::thread_id_ref_type register_thread (threads::thread_init_data &data, threads::thread_pool_base *pool, error_code &ec = throws)
```

Create a new thread using the given data.

**Return** This function will return the internal id of the newly created 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.
- 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.

**Exceptions**

- invalid_status: if the runtime system has not been started yet.

```cpp
threads::thread_id_ref_type register_thread (threads::thread_init_data &data, 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.

**Return** This function will return the internal id of the newly created 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.

**Exceptions**

- invalid_status: if the runtime system has not been started yet.
thread_id_ref_type register_work (threads::thread_init_data &data, threads::thread_pool_base *pool, error_code &ec = 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 \( hpx::\text{exception} \).

**Parameters**
- \( \text{data: [in]} \) The data to use for creating the thread.
- \( \text{pool: [in]} \) The thread pool to use for launching the work.
- \( \text{ec: [in,out]} \) This represents the error status on exit, if this is pre-initialized to \( hpx::throws \) the function will throw on error instead.

**Exceptions**
- \( \text{invalid_status: if the runtime system has not been started yet.} \)

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 \( hpx::\text{exception} \).

**Parameters**
- \( \text{data: [in]} \) The data to use for creating the thread.
- \( \text{ec: [in,out]} \) This represents the error status on exit, if this is pre-initialized to \( hpx::throws \) the function will throw on error instead.

**Exceptions**
- \( \text{invalid_status: if the runtime system has not been started yet.} \)

namespace hpx

namespace threads

namespace policies

**Functions**

\( \text{std::ostream &operator<< (std::ostream &os, scheduler_base const &scheduler)} \)

**struct scheduler_base**

\#include <scheduler_base.hpp> The scheduler_base defines the interface to be implemented by all scheduler policies

**Public Types**

typedef \( \text{std::mutex pu_mutex_type} \)

using polling_function_ptr = detail::polling_status (*)(*)

using polling_work_count_function_ptr = std::size_t (*)(*)
Public Functions

HPX_NON_COPYABLE (scheduler_base)

scheduler_base (std::size_t num_threads, char const *description = "",
   thread_queue_init_parameters thread_queue_init = {}, scheduler_mode mode = nothing_special)

virtual ~scheduler_base ()

threads::thread_pool_base *get_parent_pool () const

void set_parent_pool (threads::thread_pool_base *)

std::size_t global_to_local_thread_index (std::size_t n)

std::size_t local_to_global_thread_index (std::size_t n)

char const *get_description () const

void idle_callback (std::size_t num_thread)

void do_some_work (std::size_t)
   This function gets called by the thread-manager whenever new work has been added, allowing
   the scheduler to reactivate one or more of possibly idling OS threads

virtual void suspend (std::size_t num_thread)

virtual void resume (std::size_t num_thread)

std::size_t select_active_pu (std::unique_lock<pu_mutex_type> &l, std::size_t num_thread, bool allow_fallback = false)

std::atomic<hpx::state> &get_state (std::size_t num_thread)

std::atomic<hpx::state> const &get_state (std::size_t num_thread) const

void set_all_states (hpx::state s)

void set_all_states_at_least (hpx::state s)

bool has_reached_state (hpx::state s) const

bool is_state (hpx::state s) const

std::pair<hpx::state, hpx::state> get_minmax_state () const

scheduler_mode get_scheduler_mode () const

bool has_scheduler_mode (scheduler_mode mode) const

virtual void set_scheduler_mode (scheduler_mode mode)

void add_scheduler_mode (scheduler_mode mode)

void remove_scheduler_mode (scheduler_mode mode)

void add_remove_scheduler_mode (scheduler_mode to_add_mode, scheduler_mode to_remove_mode)

void update_scheduler_mode (scheduler_mode mode, bool set)
pu_mutex_type & get_pu_mutex (std::size_t num_thread)

std::size_t domain_from_local_thread_index (std::size_t n)

std::size_t num_domains (const std::size_t workers)

std::vector<std::size_t> domain_threads (std::size_t local_id, const std::vector<std::size_t> & ts, util::function_nonser<bool> std::size_t, std::size_t)

> pred

virtual std::int64_t get_queue_length (std::size_t num_thread = std::size_t(-1))

virtual 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 = std::size_t(-1), bool reset = false) const = 0

virtual bool is_core_idle (std::size_t num_thread) const = 0

std::int64_t get_background_thread_count ()

void increment_background_thread_count ()

void decrement_background_thread_count ()

virtual bool enumerate_threads (util::function_nonser<bool> thread_id_type > const & f, thread_schedule_state state = thread_schedule_state::unknown const = 0

virtual void abort_all_suspended_threads () = 0

virtual bool cleanup_terminated (bool delete_all) = 0

virtual bool cleanup_terminated (std::size_t num_thread, bool delete_all) = 0

virtual void create_thread (thread_init_data & data, thread_id_ref_type * id, error_code & ec) = 0

virtual bool get_next_thread (std::size_t num_thread, bool running, threads::thread_id_ref_type & thrd, bool enable_stealing) = 0

virtual void schedule_thread (threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback = false, thread_priority priority = thread_priority::normal) = 0

virtual void schedule_thread_last (threads::thread_id_ref_type thrd, threads::thread_schedule_hint schedulehint, bool allow_fallback = false, thread_priority priority = thread_priority::normal) = 0

virtual void destroy_thread (threads::thread_data * thrd) = 0

virtual bool wait_or_add_new (std::size_t num_thread, bool running, std::int64_t & idle_loop_count, bool enable_stealing, std::size_t & added) = 0
virtual void on_start_thread (std::size_t num_thread) = 0
virtual void on_stop_thread (std::size_t num_thread) = 0
virtual void on_error (std::size_t num_thread, std::exception_ptr const & e) = 0
virtual void reset_thread_distribution ()

std::ptrdiff_t get_stack_size (threads::thread_stacksize stacksize) const

void set_mpi_polling_functions (polling_function_ptr mpi_func,
                                polling_work_count_function_ptr mpi_work_count_func)

void clear_mpi_polling_function ()

void set_cuda_polling_functions (polling_function_ptr cuda_func,
                                 polling_work_count_function_ptr cuda_work_count_func)

void clear_cuda_polling_function ()

detail::polling_status custom_polling_function () const

std::size_t get_polling_work_count () const

Public Static Functions

static detail::polling_status null_polling_function ()

static std::size_t null_polling_work_count_function ()

Protected Attributes

util::cache_line_data<std::atomic<scheduler_mode>> mode_
std::vector<pu_mutex_type> suspend_mtxs_
std::vector<std::condition_variable> suspend_conds_
std::vector<pu_mutex_type> pu_mtxs_
std::vector<std::atomic<hpx::state>> states_
char const * description_
thread_queue_init_parameters thread_queue_init_
threads::thread_pool_base * parent_pool_
std::atomic<std::int64_t> background_thread_count_
std::atomic<polling_function_ptr> polling_function_mpi_
std::atomic<polling_function_ptr> polling_function_cuda_
std::atomic<polling_work_count_function_ptr> polling_work_count_function_mpi_
std::atomic<polling_work_count_function_ptr> polling_work_count_function_cuda_

namespace hpx
namespace threads

namespace policies

 Enums

 enum scheduler_mode
 This enumeration describes the possible modes of a scheduler.
 Values:

 nothing_special = 0x000
 As the name suggests, this option can be used to disable all other options.

do_background_work = 0x001
 The scheduler will periodically call a provided callback function from a special HPX thread
to enable performing background-work, for instance driving networking progress or garbage-
collect AGAS.

 reduce_thread_priority = 0x002
 The kernel priority of the os-thread driving the scheduler will be reduced below normal.

delay_exit = 0x004
 The scheduler will wait for some unspecified amount of time before exiting the scheduling loop
while being terminated to make sure no other work is being scheduled during processing the
shutdown request.

 fast_idle_mode = 0x008
 Some schedulers have the capability to act as ‘embedded’ schedulers. In this case it needs to
periodically invoke a provided callback into the outer scheduler more frequently than normal.
This option enables this behavior.

 enable_elasticity = 0x010
 This option allows for the scheduler to dynamically increase and reduce the number of pro-
cessing units it runs on. Setting this value not succeed for schedulers that do not support this
functionality.

 enable_stealing = 0x020
 This option allows schedulers that support work thread/stealing to enable/disable it

 enable_stealing_numa = 0x040
 This option allows schedulers that support it to disallow stealing between numa domains

 assign_work_round_robin = 0x080
 This option tells schedulersthat support it to add tasks round robin to queues on each core

 assign_work_thread_parent = 0x100
 This option tells schedulers that support it to add tasks round to the same core/queue that the
parent task is running on

 steal_high_priority_first = 0x200
 This option tells schedulers that support it to always (try to) steal high priority tasks from other
queues before finishing their own lower priority tasks

 steal_after_local = 0x400
 This option tells schedulers that support it to steal tasks only when their local queues are empty

 enable_idle_backoff = 0x0800
 This option allows for certain schedulers to explicitly disable exponential idle-back off
**default_mode** = do\_background\_work \ reduce\_thread\_priority \ delay\_exit \ enable\_stealing \ enable\_stealing\_numa

This option represents the default mode.

**all\_flags** = do\_background\_work \ reduce\_thread\_priority \ delay\_exit \ fast\_idle\_mode \ enable\_elasticity \ enable\_idle\_backoff

This enables all available options.

namespace hpx

**Enums**

enum state

Values:

state\_invalid = -1
state\_initialized = 0
first\_valid\_runtime\_state = state\_initialized
state\_pre\_startup = 1
state\_startup = 2
state\_pre\_main = 3
state\_starting = 4
state\_running = 5
state\_suspended = 6
state\_pre\_sleep = 7
state\_sleeping = 8
state\_pre\_shutdown = 9
state\_shutdown = 10
state\_stopping = 11
state\_terminating = 12
state\_stopped = 13
last\_valid\_runtime\_state = state\_stopped

namespace hpx

namespace threads

**Functions**

thread\_data *get\_self\_id\_data ()

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\_data *get\_thread\_id\_data (thread\_id\_ref\_type const &tid)

thread\_data *get\_thread\_id\_data (thread\_id\_type const &tid)
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()
    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()
    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).

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()
    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()
    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()
    The function get_self_stacksize_enum returns the stack size of the ./

std::uint32_t get_parent_locality_id()
    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()
    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.
class thread_data : public thread_data_reference_counting
#include <thread_data.hpp> A thread is the representation of a ParalleX thread. It’s a first class object in ParalleX. 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.

Subclassed by hpx::threads::thread_data_stackful, hpx::threads::thread_data_stackless

Public Types

using spinlock_pool = util::spinlock_pool<thread_data>

Public Functions

thread_data (thread_data const&)
thread_data (thread_data&&)
thread_data &operator= (thread_data const&)
thread_data &operator= (thread_data&&)

thread_state get_state (std::memory_order order = std::memory_order_acquire) const

The get_state function queries the state of this thread instance.

Return This function returns the current state of this thread. It will return one of the values as defined by the thread_state enumeration.

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.

thread_state set_state (thread_schedule_state state, thread_restart_state state_ex = thread_restart_state::unknown, std::memory_order load_order = std::memory_order_acquire, std::memory_order exchange_order = std::memory_order_seq_cst)

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 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
- newstate: [in] The new state to be set for the thread.

bool set_state_tagged (thread_schedule_state newstate, thread_state &prev_state, thread_state &new_tagged_state, std::memory_order exchange_order = std::memory_order_seq_cst)
bool \texttt{restore\_state} (thread\_state \texttt{new\_state}, thread\_state \texttt{old\_state}, std::memory\_order \texttt{load\_order} = \texttt{std::memory\_order\_relaxed}, std::memory\_order \texttt{load\_exchange} = \texttt{std::memory\_order\_seq\_cst})

The \texttt{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.

\textbf{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 \texttt{threadmanager::set\_state} should be used.

\textbf{Return} This function returns \texttt{true} if the state has been changed successfully

\textbf{Parameters}

- \texttt{newstate} [in] The new state to be set for the thread.
- \texttt{oldstate} [in] The old state of the thread which still has to be the current state.

bool \texttt{restore\_state} (thread\_schedule\_state \texttt{new\_state}, thread\_restart\_state \texttt{state\_ex}, thread\_state \texttt{old\_state}, std::memory\_order \texttt{load\_exchange} = \texttt{std::memory\_order\_seq\_cst})

\texttt{constexpr std::uint64\_t get\_component\_id()} const

Return the id of the component this thread is running in.

\texttt{util::thread\_description get\_description()} const

\texttt{util::thread\_description set\_description} (\texttt{util::thread\_description})

\texttt{util::thread\_description get\_lco\_description()} const

\texttt{util::thread\_description set\_lco\_description} (\texttt{util::thread\_description})

\texttt{constexpr std::uint32\_t get\_parent\_locality\_id()} const

Return the locality of the parent thread.

\texttt{constexpr thread\_id\_type get\_parent\_thread\_id()} const

Return the thread id of the parent thread.

\texttt{constexpr std::size\_t get\_parent\_thread\_phase()} const

Return the phase of the parent thread.

\texttt{constexpr util::backtrace const *get\_backtrace()} const

\texttt{util::backtrace const *set\_backtrace} (\texttt{util::backtrace const *})

\texttt{constexpr thread\_priority get\_priority()} const

\texttt{void set\_priority} (\texttt{thread\_priority priority})

\texttt{bool interruption\_requested()} const

\texttt{bool interruption\_enabled()} const

\texttt{bool set\_interruption\_enabled} (\texttt{bool enable})

\texttt{void interrupt} (\texttt{bool flag = true})

\texttt{bool interruption\_point} (\texttt{bool throw\_on\_interrupt = true})
bool add_thread_exit_callback (util::function_nonser<\void>)
    > const &f

void run_thread_exit_callbacks ()
void free_thread_exit_callbacks ()
bool is_stackless () const
void destroy_thread ()

policies::scheduler_base *get_scheduler_base () const
std::size_t get_last_worker_thread_num () const
void set_last_worker_thread_num (std::size_t last_worker_thread_num)
std::ptrdiff_t get_stack_size () const

thread_stacksize get_stack_size_enum () const

template<typename ThreadQueue>
ThreadQueue &get_queue ()

coroutine_type::result_type operator ()
    (hpx::execution_base::this_thread::detail::agent_storage
     *agent_storage)

    Execute the thread function.

Return 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.

virtual thread_id_type get_thread_id () const
virtual std::size_t get_thread_phase () const
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 ()
virtual void destroy () = 0
Protected Functions

\texttt{thread\_restart\_state set\_state\_ex (thread\_restart\_state new\_state)}

The set\_state function changes the extended state of this thread instance.

\textbf{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.

\textbf{Parameters}

- \texttt{newstate: [in]} The new extended state to be set for the thread.

\texttt{void rebind\_base (thread\_init\_data &init\_data)}

Private Members

\texttt{std::atomic<thread\_state> current\_state_}

\texttt{thread\_priority priority_}

\texttt{bool requested\_interrupt_}

\texttt{bool enabled\_interrupt_}

\texttt{bool ran\_exit\_funcs_}

\texttt{const bool is\_stackless_}

\texttt{std::forward\_list<util::function\_nonser<void () >> exit\_funcs_}

\texttt{policies::scheduler\_base *scheduler\_base_}

\texttt{std::size\_t last\_worker\_thread\_num_}

\texttt{std::ptrdiff\_t stacksize_}

\texttt{thread\_stacksize stacksize\_enum_}

\texttt{void *queue_}

namespace hpx

namespace threads

class thread\_data\_stackful : public hpx::threads::thread\_data

#include <thread\_data\_stackful.hpp> A thread is the representation of a ParalleX thread. It’s a first class object in ParalleX. In our implementation this is a user level thread running on top of one of the OS threads spawned by the thread-manager.

A \textit{thread} encapsulates:

- A thread status word (see the functions \texttt{thread::get\_state} and \texttt{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, \textit{threads} are not created or executed directly. All functionality related to the management of \textit{threads} is implemented by the thread-manager.
Public Functions

coroutine_type::result_type call (hpx::execution_base::this_thread::detail::agent_storage *agent_storage)

std::size_t get_thread_data () const

std::size_t set_thread_data (std::size_t data)

void init ()

void rebind (thread_init_data &init_data)

thread_data_stackful (thread_init_data &init_data, void *queue, std::ptrdiff_t stacksize, thread_id_addref addref)

~thread_data_stackful ()

void destroy ()

Public Static Functions

thread_data *create (thread_init_data &init_data, void *queue, std::ptrdiff_t stacksize, thread_id_addref addref = thread_id_addref::yes)

Private Functions

thread_data *this_ ()

Private Members

coroutine_type coroutine_

execution_agent agent_

Private Static Attributes

util::internal_allocator<thread_data_stackful> thread_alloc_

namespace hpx

namespace threads

class thread_data_stackless : public hpx::threads::thread_data
#include <thread_data_stackless.hpp> A thread is the representation of a ParalleX thread. It’s a first class object in ParalleX. 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 Functions**

- stackless_coroutine_type::result_type **call**()
- std::size_t **get_thread_data**() const
- std::size_t **set_thread_data**(std::size_t data)
- void **init**( )
- void **rebind**(thread_init_data &init_data)
- thread_data_stackless**(thread_init_data &init_data, void *queue, std::ptrdiff_t stacksize, thread_id_addref addref)**
- ~thread_data_stackless**( )
- void **destroy**( )

**Public Static Functions**

- thread_data *create**(thread_init_data &data, void *queue, std::ptrdiff_t stacksize, thread_id_addref addref = thread_id_addref::yes)

**Private Functions**

- thread_data *this_*()

**Private Members**

- stackless_coroutine_type **coroutine_**

**Private Static Attributes**

- util::internal_allocator<thread_data_stackless> **thread_alloc_**

namespace hpx

namespace threads
Functions

util::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.

Return 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>”.

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.

util::thread_description set_thread_description (thread_id_type const &id, util::thread_description const &desc = util::thread_description(), error_code &ec = throws)

util::thread_description get_thread_lco_description (thread_id_type const &id, error_code &ec = throws)

util::thread_description set_thread_lco_description (thread_id_type const &id, util::thread_description const &desc = util::thread_description(), error_code &ec = throws)

namespace util

Functions

std::ostream &operator<< (std::ostream&, thread_description const&)

std::string as_string (thread_description const &desc)

struct thread_description

Public Types

enum data_type

Values:

data_type_description = 0

data_type_address = 1
Public Functions

thread_description()
constexpr thread_description(char const*)

template<typename F, typename = typename std::enable_if<!std::is_same<F, thread_description>::value && !traits::is_action<F>::value>::type>
constexpr thread_description(F const&, char const* = nullptr)
	
template<typename Action, typename = typename std::enable_if<traits::is_action<Action>::value>::type>
constexpr thread_description(Action, char const* = nullptr)

constexpr data_type kind() const
constexpr char const* get_description() const
constexpr std::size_t get_address() const
constexpr operator bool() const
constexpr bool valid() const

Private Functions

void init_from_alternative_name(char const* altname)

namespace hpx

namespace this_thread

Functions

threads::thread_restart_state suspend(threads::thread_schedule_state state, threads::thread_id_type id, util::thread_description const &description = util::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.

Exceptions
• 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::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::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::invalid_status.
threads::thread_restart_state suspends(threads::thread_schedule_state state = threads::thread_schedule_state::pending,
util::thread_description const &description =
util::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.

**Exceptions**
- 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::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::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::invalid_status.

threads::thread_restart_state suspend(hpx::chrono::steady_time_point const &abs_time,
threads::thread_id_type id,
util::thread_description const &description =
util::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.

**Exceptions**
- 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::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::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::invalid_status.

threads::thread_restart_state suspend(hpx::chrono::steady_time_point const &abs_time,
util::thread_description const &description =
util::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.

**Exceptions**
- 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::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::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::invalid_status.

threads::thread_restart_state suspend(hpx::chrono::steady_duration const &rel_time,
util::thread_description const &description =
util::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 thread after the given duration.

**Note** Must be called from within a HPX-thread.

**Exceptions**

- 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::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::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::invalid_status`.

```cpp
treads::thread_restart_state suspend(hpx::chrono::steady_duration const &rel_time,
threads::thread_id_type const &id,
util::thread_description const &description =
util::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 thread after the given duration.

**Note** Must be called from within a HPX-thread.

**Exceptions**

- 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::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::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::invalid_status`.

```cpp
treads::thread_restart_state suspend(std::uint64_t ms,
util::thread_description const &description =
util::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 thread after the given time (specified in milliseconds).

**Note** Must be called from within a HPX-thread.

**Exceptions**

- 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::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::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::invalid_status`.

```cpp
threads::thread_pool_base *get_pool(error_code &ec = throws)
```

Returns a pointer to the pool that was used to run the current thread.

**Exceptions**

- 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
**namespace threads**

### Functions

- **thread_state**
  ```cpp
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.

  **Return** 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.

  **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:
  - 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.

- **thread_id_ref_type**
  ```cpp
  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 a timer to set the state of the given thread to the given new value after it expired (at the given time)

  **Return**
  **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:
• 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.

\[
\text{thread_id_ref_type set_thread_state} (\text{thread_id_type } \text{const } \&\text{id}, \\
\text{hpx::chrono::steady_time_point } \text{const } \&\text{abs_time}, \text{ thread_schedule_state } \text{state } = \text{thread_schedule_state::pending, thread_restart_state} \text{stateex } = \text{thread_restart_state::timeout, thread_priority} \text{priority } = \text{thread_priority::normal, bool retry_on_active} \text{ = true, error_code}& = \text{throws})
\]

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)

Return

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:

\[
\text{thread_id_ref_type set_thread_state} (\text{thread_id_type } \text{const } \&\text{id}, \\
\text{hpx::chrono::steady_duration } \text{const } \&\text{rel_time}, \text{ thread_schedule_state } \text{state } = \text{thread_schedule_state::pending, thread_restart_state} \text{stateex } = \text{thread_restart_state::timeout, thread_priority} \text{priority } = \text{thread_priority::normal, bool retry_on_active} \text{ = true, error_code & ec = throws})
\]

\[
\text{the\_id\_ref\_type get_thread_state} (\text{thread_id_type } \text{const } \&\text{id}, \text{error_code } \text{const } \&\text{ec } = \text{throws})
\]

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).

Return 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.

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.

Return 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.

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.

Parameters

• 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.

`std::size_t get_thread_phase(thread_id_type const &id, error_code &ec = throws)`

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.

**Return** 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.

**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.

`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.

**Return** This function returns `true` if the given thread can be interrupted at this point in time. It will return `false` 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**

- `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.

`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.

**Return** This function returns the previous value of whether the given thread could have been interrupted.

**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.

`bool get_thread_interruption_requested(thread_id_type const &id, error_code &ec = throws)`

Returns whether the given thread has been flagged for interruption.

**Return** This function returns `true` if the given thread was flagged for interruption. It will return `false` 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**

- `id`: [in] The thread id of the thread which should be queried.
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.

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)
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)
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

Exceptions
• 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::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::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::invalid_status.

namespace hpx

namespace threads

class thread_init_data

Public Functions
	hread_init_data ()

thread_init_data &operator= (thread_init_data &&rhs)

thread_init_data (thread_init_data &&rhs)

template<typename F>
thread_init_data (F &&f, util::thread_description const &desc, thread_priority priority_ = thread_priority::normal, thread_schedule_hint os_thread = thread_schedule_hint(), thread_stacksize stacksize_ = thread_stacksize::default_, thread_schedule_state initial_state_ = thread_schedule_state::pending, bool run_now_ = false, policies::scheduler_base *scheduler_base_ = nullptr)

Public Members

threads::thread_function_type func
thread_priority priority
thread_schedule_hint schedulehint
thread_stacksize stacksize
thread_schedule_state initial_state
bool run_now
policies::scheduler_base *scheduler_base

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.  
**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.

```cpp
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).
```

```cpp
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.

```cpp
namespace hpx

namespace threads

Functions
```

```cpp
std::ostream &operator<<(std::ostream &os, thread_pool_base const &thread_pool)

struct executor_statistics
    #include <thread_pool_base.hpp> Data structure which stores statistics collected by an executor instance.

Public Functions
```

```cpp
executor_statistics()
```
Public Members

std::uint64_t tasks_scheduled_
std::uint64_t tasks_completed_
std::uint64_t queue_length_

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.

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.

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.

Exceptions
• hpx::exception: if called from an HPX thread which is running on the pool itself.
**Public Functions**

```cpp
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 = 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**

```cpp
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_
```

```cpp
namespace hpx

namespace threads

namespace policies

struct thread_queue_init_parameters
```
Public Functions

`thread_queue_init_parameters(std::int64_t max_thread_count = std::int64_t(HPX_THREAD_QUEUE_MAX_THREAD_COUNT),
std::int64_t min_tasks_to_steal_pending = std::int64_t(HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_PENDING),
std::int64_t min_tasks_to_steal_staged = std::int64_t(HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_STAGED),
std::int64_t min_add_new_count = std::int64_t(HPX_THREAD_QUEUE_MIN_ADD_NEW_COUNT),
std::int64_t max_add_new_count = std::int64_t(HPX_THREAD_QUEUE_MAX_ADD_NEW_COUNT),
std::int64_t min_delete_count = std::int64_t(HPX_THREAD_QUEUE_MIN_DELETE_COUNT),
std::int64_t max_delete_count = std::int64_t(HPX_THREAD_QUEUE_MAX_DELETE_COUNT),
std::int64_t max_terminated_threads = std::int64_t(HPX_THREAD_QUEUE_MAX_TERMINATED_THREADS),
std::int64_t init_threads_count = std::int64_t(HPX_THREAD_QUEUE_INIT_THREADS_COUNT),
double max_idle_backoff_time = double(HPX_IDLE_BACKOFF_TIME_MAX),
std::ptrdiff_t small_stacksize = HPX_SMALL_STACK_SIZE,
std::ptrdiff_t medium_stacksize = HPX_MEDIUM_STACK_SIZE,
std::ptrdiff_t large_stacksize = HPX_LARGE_STACK_SIZE,
std::ptrdiff_t huge_stacksize = HPX_HUGE_STACK_SIZE)`

Public Members

`std::int64_t max_thread_count_
std::int64_t min_tasks_to_steal_pending_
std::int64_t min_tasks_to_steal_staged_
std::int64_t min_add_new_count_
std::int64_t max_add_new_count_
std::int64_t min_delete_count_
std::int64_t max_delete_count_
std::int64_t max_terminated_threads_
std::int64_t init_threads_count_
double max_idle_backoff_time_
const std::ptrdiff_t small_stacksize_
const std::ptrdiff_t medium_stacksize_
const std::ptrdiff_t large_stacksize_
const std::ptrdiff_t huge_stacksize_`
typedef std::ptrdiff_t nostack_stacksize_

threadmanager

The contents of this module can be included with the header hpx/modules/threadmanager.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/threadmanager.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

amespace threads

class threadmanager

#include <threadmanager.hpp> The thread-manager class is the central instance of management for all (non-depleted) threads

Public Types

typedef threads::policies::callback_notifier notification_policy_type
typedef std::unique_ptr<thread_pool_base> pool_type
typedef threads::policies::scheduler_base scheduler_type
typedef std::vector<pool_type> pool_vector

Public Functions

threadmanager(hpx::util::runtime_configuration &rtcfg_, notification_policy_type &notifier, detail::network_background_callback_type network_background_callback = detail::network_background_callback_type())

~threadmanager()
void init()
void create_pools()
void print_pools(std::ostream&)
FIXME move to private and add hpx:printpools cmd line option.

thread_pool_base &default_pool() const
scheduler_type &default_scheduler() 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(const std::size_t pool_index)

The function `pool_exists` checks if a pool exists with the given index. It returns `true` if the pool exists, otherwise `false`.

### thread_id_ref_type register_work(thread_init_data &data, error_code &ec = throws)

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**
- `func`: [in] The function or function object to execute as the thread’s function. This must have a signature as defined by `thread_function_type`.
- `description`: [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.

### void register_thread(thread_init_data &data, thread_id_ref_type &id, error_code &ec = throws)

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**
- `func`: [in] The function or function object to execute as the thread’s function. This must have a signature as defined by `thread_function_type`.
- `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.
- `description`: [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.

### bool run()

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`.

**Return** The function returns `true` if the thread manager has been started successfully, otherwise it returns `false`.

### void stop(bool blocking = true)

Forcefully stop the thread-manager.

**Parameters**
- `blocking`: `bool` is_busy() 
- `is_busy`: `bool` is_idle() 
- `is_idle`: `void` wait() 
- `wait`: `void` suspend() 
- `suspend`: `void` resume() 
- `resume`: `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 = std::size_t(-1), bool reset = false)

return the number of HPX-threads with the given state

Note This function lock the internal OS lock in the thread manager

std::int64_t get_idle_core_count()

mask_type get_idle_core_mask()

std::int64_t get_background_thread_count()

bool enumerate_threads(util::function_nonser<bool>
thread_id_type
> const &f, thread_schedule_state state = thread_schedule_state::unknown const

void abort_all_suspended_threads()

bool cleanup_terminated(bool delete_all)

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)

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(const std::string &pool_name) const

void set_scheduler_mode(threads::policies::scheduler_mode mode)

void add_scheduler_mode(threads::policies::scheduler_mode mode)

void add_removeScheduler_mode(threads::policies::scheduler_mode
     to_add_mode, threads::policies::scheduler_mode
     to_remove_mode)

void remove_scheduler_mode(threads::policies::scheduler_mode mode)

void reset_thread_distribution()

void init_tss(std::size_t global_thread_num)

void deinit_tss()

std::int64_t get_queue_length(bool reset)

std::int64_t get_cumulative_duration(bool reset)
\begin{verbatim}
std::int64_t get_thread_count_unknown (bool reset)
std::int64_t get_thread_count_active (bool reset)
std::int64_t get_thread_count_pending (bool reset)
std::int64_t get_thread_count_suspended (bool reset)
std::int64_t get_thread_count_terminated (bool reset)
std::int64_t get_thread_count_staged (bool reset)

Private Types

typedef std::mutex mutex_type

Private Members

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_

\end{verbatim}

\textit{timed_execution}

The contents of this module can be included with the header \texttt{hpx/modules/timed_execution.hpp}. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we \textit{strongly} suggest only including the module header \texttt{hpx/modules/timed_execution.hpp}, not the particular header in which the functionality you would like to use is defined. See \textit{Public API} for a list of names 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
\end{verbatim}
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.

Return f(ts...)'s result through a future

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.

Parameters

- exec: [in] The executor object to use for scheduling of the function f.
- rel_time: [in] The duration of time after which the given function should be scheduled to run.
- f: [in] The function which will be scheduled using the given executor.
- ts...: [in] Additional arguments to use to invoke f.

Private Functions

template<typename Executor, typename F, typename ...Ts>
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.

Return f(ts...)'s result through a future

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.

Parameters

- exec: [in] The executor object to use for scheduling of the function f.
- abs_time: [in] The point in time the given function should be scheduled at to run.
- f: [in] The function which will be scheduled using the given executor.
- ts...: [in] Additional arguments to use to invoke f.

Private Functions

template<typename Executor, typename F, typename ...Ts>
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.

**Parameters**
- `exec`: [in] The executor object to use for scheduling of the function f.
- `rel_time`: [in] The duration of time after which the given function should be scheduled to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts...`: [in] Additional arguments to use to invoke f.

**Private Functions**

```cpp
template<typename Executor, typename F, typename ...Ts>
declaytype(auto) friend tag fallback_invoke(post_after_t, Executor &exec, hpx::chrono::steady_duration const &rel_time, F &&f, Ts&&... ts)
```

#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.

**Parameters**
- `exec`: [in] The executor object to use for scheduling of the function f.
- `abs_time`: [in] The point in time the given function should be scheduled at to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts...`: [in] Additional arguments to use to invoke f.

**Private Functions**

```cpp
template<typename Executor, typename F, typename ...Ts>
declaytype(auto) friend tag fallback_invoke(post_at_t, Executor &exec, hpx::chrono::steady_time_point const &abs_time, F &&f, Ts&&... ts)
```

#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.

**Return** f(ts...)'s result
**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.

**Parameters**
- `exec`: [in] The executor object to use for scheduling of the function `f`.
- `rel_time`: [in] The duration of time after which the given function should be scheduled to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts...`: [in] Additional arguments to use to invoke `f`.

**Private Functions**

```cpp
template<typename Executor, typename F, typename ...Ts>
decltype(auto) friend tag_fallback_invoke(sync_execute_after_t, Executor &&exec, hpx::chrono::steady_duration const &rel_time, F &&f, Ts&&... ts)
```

This synchronously creates a single function invocation `f()` using the associated executor at the given point in time.

**Return** `f(ts...)`’s result

**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.

**Parameters**
- `exec`: [in] The executor object to use for scheduling of the function `f`.
- `abs_time`: [in] The point in time the given function should be scheduled at to run.
- `f`: [in] The function which will be scheduled using the given executor.
- `ts...`: [in] Additional arguments to use to invoke `f`.

**Private Functions**

```cpp
namespace hpx
```

```cpp
namespace parallel
```

```cpp
namespace execution
```
**Typedefs**

```cpp
typedef sequenced_timed_executor = timed_executor<hpx::execution::sequenced_executor>
typedef parallel_timed_executor = timed_executor<hpx::execution::parallel_executor>
```

**Public Types**

```cpp
typedef std::decay<BaseExecutor>::type base_executor_type
typedef hpx::traits::executor_execution_category<base_executor_type>::type execution_category
typedef hpx::traits::executor_parameters_type<base_executor_type>::type parameters_type
```

**Public Functions**

```cpp
timed_executor(hpx::chrono::steady_time_point const &abs_time)
timed_executor(hpx::chrono::steady_duration const &rel_time)
timed_executor(Executor &&exec, hpx::chrono::steady_time_point const &abs_time)
timed_executor(Executor &&exec, hpx::chrono::steady_duration const &rel_time)
timed_executor(F&&, typename ...Ts> hpx::util::detail::invoke_deferred_result<F, Ts...>::type sync_execute (F &&f, Ts&&... ts)
timed_executor(F&&, typename ...Ts> hpx::future<typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute (F &&f, Ts&&... ts)
void post (F &&f, Ts&&... ts)
```

**Public Members**

```cpp
BaseExecutor exec_
std::chrono::steady_clock::time_point execute_at_
```

```cpp
namespace hpx
namespace parallel
namespace execution
```
**Typedefs**

```cpp
template<typename T>
using is_timed_executor_t = typename is_timed_executor<T>::type
```

**Variables**

```cpp
template<typename T> HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::parallel::execution::is_timed_executor_v = is_timed_executor<T>::value
```

timing

The contents of this module can be included with the header `hpx/modules/timing.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/timing.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace chrono

struct high_resolution_clock

Public Static Functions

```cpp
static std::uint64_t now()
static std::uint64_t min()
static std::uint64_t max()
```

```cpp
namespace util

**Typedefs**

typedef hpx::chrono::steady_duration instead
```

```cpp
namespace hpx

namespace chrono

class high_resolution_timer
```
Public Functions

high_resolution_timer()

high_resolution_timer(double t)

void restart()

double elapsed() const

std::int64_t elapsed_microseconds() const

std::int64_t elapsed_nanoseconds() const

double elapsed_max() const

double elapsed_min() const

Public Static Functions

static double now()

Protected Static Functions

static std::uint64_t take_time_stamp()

Private Members

std::uint64_t start_time_

namespace hpx

namespace util

// Additional content for scoped_timer struct...

Public Functions

scoped_timer(T &t, bool enabled = true)

scoped_timer(scoped_timer const&)

scoped_timer(scoped_timer &&rhs)

~scoped_timer()

scoped_timer &operator=(scoped_timer const &rhs)

scoped_timer &operator=(scoped_timer &&rhs)

bool enabled() const
Private Members

std::uint64_t started_at_
T *t_

namespace hpx

namespace chrono

class steady_duration

Public Functions

steady_duration (value_type const &rel_time)

template<typename Rep, typename Period>
steady_duration (std::chrono::duration<Rep, Period> const &rel_time)

value_type const &value () const

steady_clock::time_point from_now () const

Private Types

typedef steady_clock::duration value_type

Private Members

value_type _rel_time

class steady_time_point

Public Functions

steady_time_point (value_type const &abs_time)

template<typename Clock, typename Duration>
steady_time_point (std::chrono::time_point<Clock, Duration> const &abs_time)

value_type const &value () const

Private Types

typedef steady_clock::time_point value_type
Private Members

`value_type abs_time`

namespace hpx

namespace util

class tick_counter

Public Functions

tick_counter (std::uint64_t &output)
~tick_counter()

Protected Static Functions

static std::uint64_t take_time_stamp()

Private Members

const std::uint64_t start_time_
std::uint64_t &output_

topology

The contents of this module can be included with the header hpx/modules/topology.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/topology.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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

```c
enum hpx_hwloc_membind_policy
```

Please see hwloc documentation for the corresponding enums HWLOC_MEMBIND_XXX.

Values:

```c
membind_default = HWLOC_MEMBIND_DEFAULT
membind_firsttouch = HWLOC_MEMBIND_FIRSTTOUCH
membind_bind = HWLOC_MEMBIND_BIND
membind_interleave = HWLOC_MEMBIND_INTERLEAVE
membind_replicate = HWLOC_MEMBIND_REPLICATE
membind_nexttouch = HWLOC_MEMBIND_NEXTTOUCH
membind_mixed = HWLOC_MEMBIND_MIXED
membind_user = HWLOC_MEMBIND_MIXED + 256
```

Functions

```c
topology & create_topology()
HPX_NODISCARD unsigned int hpx::threads::hardware_concurrency()
std::size_t get_memory_page_size()
```

```c
struct hpx_hwloc_bitmap_wrapper
```

Public Functions

```c
HPX_NON_COPYABLE (hpx_hwloc_bitmap_wrapper)
```

```c
hpx_hwloc_bitmap_wrapper()
```

```c
hpx_hwloc_bitmap_wrapper (void *bmp)
```

```c
~hpx_hwloc_bitmap_wrapper()
```

```c
void reset (hwloc_bitmap_t bmp)
```

```c
operator bool () const
```

```c
hwloc_bitmap_t get_bmp () const
```

Private Members

```c
hwloc_bitmap_t bmp_
```
**Friends**

```cpp
std::ostream &operator<<(std::ostream &os, hpx_hwloc_bitmap_wrapper const *bmp)
```

**struct topology**

### Public Functions

**topology()**

**~topology()**

```cpp
std::size_t get_socket_number(std::size_t num_thread, error_code & ec = throws) const
```

Return the Socket number of the processing unit the given thread is running on.

**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.

```cpp
std::size_t get_numa_node_number(std::size_t num_thread, error_code & ec = throws) const
```

Return the NUMA node number of the processing unit the given thread is running on.

**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.

```cpp
mask_cref_type get_machine_affinity_mask(error_code & ec = throws) const
```

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.

```cpp
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.

```cpp
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**

- `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
• 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_numa_node_affinity_mask_from_numa_node (std::size_t num_node) const
Return a bit mask where each set bit corresponds to a processing unit associated with the given NUMA node.

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_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
• 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
• 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
• 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 *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
• 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 \texttt{print\_affinity\_mask}\ (std::ostream &os, std::size\_t num\_thread, mask\_cref\_type m, 
\hspace{1em} const std::string &pool\_name) const

Prints the.

\textbf{Parameters}
\begin{itemize}
  \item \texttt{m}: to os in a human readable form
\end{itemize}

bool \texttt{reduce\_thread\_priority}\ (error\_code &ec = \texttt{throws}) const

Reduce thread priority of the current thread.

\textbf{Parameters}
\begin{itemize}
  \item \texttt{ec}: [in,out] this represents the error status on exit, if this is pre-initialized to hpx::\texttt{throws} the function will throw on error instead.
\end{itemize}

\texttt{std::size\_t get\_number\_of\_sockets}() const

Return the number of available NUMA domains.

\texttt{std::size\_t get\_number\_of\_numa\_nodes}() const

Return the number of available NUMA domains.

\texttt{std::size\_t get\_number\_of\_cores}() const

Return the number of available cores.

\texttt{std::size\_t get\_number\_of\_pus}() const

Return the number of available hardware processing units.

\texttt{std::size\_t get\_number\_of\_numa\_node\_cores}(std::size\_t numa) const

Return number of cores in given numa domain.

\texttt{std::size\_t get\_number\_of\_numa\_node\_pus}(std::size\_t numa) const

Return number of processing units in a given numa domain.

\texttt{std::size\_t get\_number\_of\_socket\_pus}(std::size\_t socket) const

Return number of processing units in a given socket.

\texttt{std::size\_t get\_number\_of\_core\_pus}(std::size\_t core) const

Return number of processing units in given core.

\texttt{std::size\_t get\_number\_of\_socket\_cores}(std::size\_t socket) const

Return number of cores units in given socket.

\texttt{std::size\_t get\_core\_number}(std::size\_t num\_thread, error\_code& = \texttt{throws}) const

\texttt{std::size\_t get\_pu\_number}(std::size\_t num\_core, std::size\_t num\_pu, error\_code &ec = \texttt{throws}) const

\texttt{mask\_type get\_cpubind\_mask}(error\_code &ec = \texttt{throws}) const

\texttt{mask\_type get\_cpubind\_mask}(std::thread &handle, error\_code &ec = \texttt{throws}) const

\texttt{hwloc\_bitmap\_ptr cpuset\_to\_nodeset}(mask\_cref\_type cpuset) const

convert a cpu mask into a numa node mask in hwloc bitmap form

void \texttt{write\_to\_log}() const

void *\texttt{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, 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(const void *addr, std::size_t len) const

bool set_area_membind_nodeset(const void *addr, std::size_t len, void *nodeset) const

int get_numa_domain(const void *addr) const

void deallocate(void *addr, std::size_t len) const
Free memory that was previously allocated by allocate.

void print_vector(std::ostream &os, std::vector<std::size_t> const &v) const
void print_mask_vector(std::ostream &os, std::vector<mask_type> const &v) const
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

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)

std::size_t init_socket_number(std::size_t num_thread)

std::size_t init_numa_node_number(std::size_t num_thread)

std::size_t init_core_number(std::size_t num_thread)

void extract_node_mask(hwloc_obj_t parent, mask_type &mask) const

std::size_t extract_node_count(hwloc_obj_t parent, hwloc_obj_type_t type, std::size_t count) const
mask_type init_machine_affinity_mask() const
mask_type init_socket_affinity_mask(std::size_t num_thread) const
mask_type init_numa_node_affinity_mask(std::size_t num_thread) const
mask_type init_core_affinity_mask(std::size_t num_thread) const
void init_num_of_pus()

Private Members

hwloc_topology_t topo
std::size_t num_of_pus_
bool use_pus_as_cores_
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_
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

mask_type empty_mask
std::size_t memory_page_size_
constexpr std::size_t pu_offset = 0
constexpr std::size_t core_offset = 0

Friends

std::size_t get_memory_page_size()
type_support

The contents of this module can be included with the header \texttt{hpx/modules/type_support.hpp}. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header \texttt{hpx/modules/type_support.hpp}, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace util

**Typedefs**

```cpp
template<typename ...T>
using always_void_t = typename always_void<T...>::type

template<typename ...T>
struct always_void
```

**Public Types**

```cpp
template<>
using type = void
```

namespace hpx

namespace util

**Typedefs**

```cpp
template<typename T>
using decay_unwrap_t = typename decay_unwrap<T>::type
```

namespace hpx

namespace util

**Typedefs**

```cpp
template<template<typename...> class Op, typename ...
Args>
using is_detected = typename detail::detector<nonesuch, void, Op, Args...>::value_t

template<template<typename...> class Op, typename ...
Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type

template<typename Default, template<typename...> class Op, typename ...
Args>
using detected_or = detail::detector<Default, void, Op, Args...>

template<typename Default, template<typename...> class Op, typename ...
Args>
using detected_or_t = typename detected_or<Default, Op, Args...>::type

template<typename Expected, template<typename...> class Op, typename ...
Args>
```
using is_detected_exact = std::is_same<Expected, detected_t<Op, Args...>>

template<typename To, template<typename...> class Op, typename ...Args>
using is_detected_convertible = std::is_convertible<detected_t<Op, Args...>, To>

struct nonesuch

    Public Functions

    nonesuch()
    ~nonesuch()
    nonesuch(nonesuch const&)
    void operator=(nonesuch const&)

namespace hpx

    namespace util

        template<typename T>
        struct identity

        Public Types

        template<>
        using type = T

namespace hpx

    namespace util

        // Typedefs

        template<bool Enable, typename C1, typename C2>
        using lazy_conditional_t = typename lazy_conditional<Enable, C1, C2>::type

namespace hpx

    namespace util

        template<typename T>
        struct lazy_enable_if<true, T>
Public Types

template<>
using type = typename T::type

namespace hpx

namespace util

Typedefs

template< std::size_t... Is >
using index_pack = pack_c< std::size_t, Is... >
template< std::size_t N >
using make_index_pack_t = typename make_index_pack< N >::type
template< std::size_t I, typename ... Ts >
using at_index_t = typename at_index< I, Ts... >::type

Variables

template< typename... Ts > HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::util::all_of_v = all_of<Ts...>::value
template< typename... Ts > HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::util::any_of_v = any_of<Ts...>::value
template< typename... Ts > HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::util::none_of_v = none_of<Ts...>::value
template< typename ... Ts >
struct pack

Public Types

typedef pack type

Public Static Attributes

constexpr std::size_t size = sizeof...(Ts)

template< typename T, T... Vs >
struct pack_c
  Subclassed by hpx::util::detail::make_index_pack_join< index_pack< Left... >, index_pack< Right... >, hpx::util::make_index_pack< 1 > >
Public Types

typedef pack_c type

Public Static Attributes

constexpr std::size_t size = sizeof...(Vs)

Defines

HPX_EXPORT_STATIC_
namespace hpx

namespace util

template<typename T, typename Tag = T>
struct static_

Public Types

typedef T value_type

typedef T &reference

typedef T const &const_reference

Public Functions

HPX_NON_COPYABLE (static_)
static_()

operator reference ()

operator const_reference () const

reference get ()

const_reference get () const

Private Types

typedef std::add_pointer<value_type>::type pointer

typedef std::aligned_storage<sizeof(value_type), std::alignment_of<value_type>::value>::type storage_type
Private Static Functions

static pointer get_address()

Private Static Attributes

static_<T, Tag>::storage_type data_
std::once_flag constructed_

struct default_constructor

Public Static Functions

template<>  
static void construct()

struct destructor

Public Functions

template<>  
~destructor()

Defines

HPX_UNUSED(x)

HPX_MAYBE_UNUSED

namespace hpx

namespace util

Variables

constexpr unused_type unused = unused_type()

struct unused_type

Public Functions

constexpr unused_type()

constexpr unused_type(unused_type const&)

constexpr unused_type(unused_type&&)

template<typename T>
constexpr unused_type(T const&)

Chapter 2. What’s so special about HPX?
template<typename T>
constexpr unused_type const &operator= (T const&) const

template<typename T>
unused_type &operator= (T const&)
constexpr unused_type const &operator= (unused_type const&) const
unused_type &operator= (unused_type const&)
constexpr unused_type const &operator= (unused_type&&) const
unused_type &operator= (unused_type&&)

template<typename T>
struct unwrap_reference<std::reference_wrapper<T>>

Public Types

typedef T type

namespace hpx

namespace util

Functions

template<typename T>
unwrap_reference<T>::type &unwrap_ref (T &t)

template<typename T>
struct unwrap_reference

Public Types

typedef T type

namespace hpx

namespace util

2.8. API reference
Public Types

typedef T type

template<typename T>
struct unwrap_reference<std::reference_wrapper<T> const>

Public Types

typedef T type

namespace hpx

namespace util

template<>
struct void_guard<void>

Public Functions

template<typename T>
void operator, (T const&) const

util

The contents of this module can be included with the header hpx/modules/util.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/util.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace util

Functions

std::size_t calculate_fanout (std::size_t size, std::size_t local_fanout)
Functions

```cpp
std::uint64_t get_and_reset_value (std::uint64_t & value, bool reset)
std::int64_t get_and_reset_value (std::int64_t & value, bool reset)
```

```cpp
template<typename T>
T get_and_reset_value (std::atomic<T> & value, bool reset)
```

```cpp
std::vector<std::int64_t> get_and_reset_value (std::vector<std::int64_t> & value, bool reset)
```

namespace hpx

namespace util

Functions

```cpp
template<typename DestType, typename Config, typename std::enable_if<!std::is_same<DestType, std::string>::value>::type = false>
DestType get_entry_as (Config const & config, std::string const & key, DestType const & dflt)
```

namespace hpx

namespace util

Functions

```cpp
template<typename Iterator>
bool insert_checked (std::pair<Iterator, bool> const & r)
```

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.

**Return** This function returns `r.second`.

**Parameters**

- `r`: [in] The return value of a std::map insert operation.

```cpp
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.

**Return** This function returns `r.second`.

**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`.

namespace hpx
namespace util

class ios_flags_saver

Public Types

typedef ::std::ios_base state_type
typedef ::std::ios_base::fmtflags aspect_type

Public Functions

ios_flags_saver (state_type &s)
ios_flags_saver (state_type &s, aspect_type const &a)
~ios_flags_saver ()
ios_flags_saver (ios_flags_saver const&)
ios_flags_saver &operator= (ios_flags_saver const&)
void restore ()

Private Members

state_type &s_save_
const aspect_type a_save_

namespace hpx

namespace util

struct manage_config

Public Types

typedef std::map<std::string, std::string> map_type

Public Functions

manage_config (std::vector<std::string> const &cfg)
void add (std::vector<std::string> const &cfg)
template<typename T>
T get_value (std::string const &key, T dflt = T()) const
Public Members

map_type config_

namespace hpx

namespace util

Functions

std::string regex_from_pattern (std::string const &pattern, error_code &ec = throws)

namespace hpx

namespace util

Functions

bool parse_sed_expression (std::string const &input, std::string &search, std::string &replace)

Parse a sed command.

Return  true if the parsing was successful, false otherwise.
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.
  • search: [out] If the parsing is successful, this string is set to the replace expression.

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 const &replace)

sed_transform (std::string const &expression)

std::string operator () (std::string const &input) const

operator bool () const

bool operator! () const
Private Members

`std::shared_ptr<command> command_`

version

The contents of this module can be included with the header `hpx/modules/version.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/version.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

Functions

`std::uint8_t major_version()`  
`std::uint8_t minor_version()`  
`std::uint8_t subminor_version()`  
`std::uint32_t full_version()`  
`std::string full_version_as_string()`  
`std::uint8_t agas_version()`  
`std::string tag()`  
`std::string full_build_string()`  
`std::string build_string()`  
`std::string boost_version()`  
`std::string boost_platform()`  
`std::string boost_compiler()`  
`std::string boost_stdlib()`  
`std::string copyright()`  
`std::string complete_version()`  
`std::string build_type()`  
`std::string build_date_time()`  
`std::string configuration_string()`
actions

The contents of this module can be included with the header `hpx/modules/actions.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/actions.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
{

    Functions

    bool is_pre_startup()

}
```

actions_base

The contents of this module can be included with the header `hpx/modules/actions_base.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/actions_base.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
{

    namespace actions

        Functions

            template<typename Action>
            threads::thread_priority action_priority()

        Definitions

            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 ‘:’.

```cpp
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server :
        public hpx::components::component_base
```
```cpp
{  
    void print_greeting ()
    {
        hpx::cout << "Hey, how are you?\n" << hpx::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 ex-
actly 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_DEFINEPlain_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.

```cpp
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

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.

```cpp
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() const
    {
      hpx::cout << "Hey, how are you?\n" << hpx::flush;
    }

    // Component actions need to be declared, this also defines the
    // type 'print_greeting_action' representing the action.

    // (continues on next page)
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.

Defines

HPX_DEFINE_PLAIN_ACTION(...)
Defines a plain action type.

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.

```cpp
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)
```

**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_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 an 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 traits

    template<
type Action, typename Enable = void>
    struct action_continuation

    Public Types

    template<>
    using type = hpx::actions::typed_continuation<
type Action::local_result_type, typename Action::remote_result_type>

namespace hpx

namespace traits

    template<
type Action, typename Enable = void>
    struct action_decorate_continuation


Public Types

```cpp
template<>
using continuation_type = typename hpx::traits::action_continuation<Action>::type
```

Public Static Functions

```cpp
static constexpr bool call (continuation_type&)
```

```cpp
namespace hpx
```

```cpp
namespace traits
```

```cpp
template<typename Action, typename Enable = void>
struct action_does_termination_detection
```

Public Static Functions

```cpp
static constexpr bool call ()
```

```cpp
namespace hpx
```

```cpp
namespace traits
```

```cpp
template<typename Action, typename Enable = void>
struct action_is_target_valid
```

Public Static Functions

```cpp
static bool call (naming::id_type const &id)
```

```cpp
namespace hpx
```

```cpp
namespace traits
```

```cpp
template<typename Action, typename Enable = void>
struct action_priority
```

Public Static Attributes

```cpp
constexpr threads::thread_priority value = threads::thread_priority::default_
```

```cpp
namespace hpx
```

```cpp
namespace traits
```

```cpp
template<typename Action, typename Enable = void>
```
struct action_schedule_thread

Public Static Functions

static void call (naming::address_type lva, naming::component_type comptype,
    threads::thread_init_data &data)

namespace hpx

namespace traits

template<typename Action, typename Enable = void>
struct action_select_direct_execution

Public Static Functions

static constexpr launch call (launch policy, naming::address_type lva)

namespace hpx

namespace traits

template<typename Action, typename Enable = void>
struct action_stacksize

Public Static Attributes

constexpr threads::thread_stacksize value = threads::thread_stacksize::default_

namespace hpx

namespace traits

template<typename Continuation, typename Enable = void>
struct action_trigger_continuation

Public Static Functions

template<typename F, typename ...Ts>
static void call (Continuation&&, F&&, Ts&&...)

namespace hpx

namespace traits

template<typename Action, typename Enable = void>
struct action_was_object_migrated

Chapter 2. What’s so special about HPX?
Public Static Functions

```cpp
static std::pair<bool, components::pinned_ptr> call (hpx::naming::gid_type const &id,
naming::address_type lva)
```

```cpp
static std::pair<bool, components::pinned_ptr> call (hpx::naming::id_type const &id,
naming::address_type lva)
```

namespace hpx

namespace traits

```cpp
template<typename T, typename Enable = void>
struct extract_action
```

Public Types

```cpp
template<>
using type = typename Action::derived_type
```

```cpp
template<>
using result_type = typename type::result_type
```

```cpp
template<>
using local_result_type = typename type::local_result_type
```

```cpp
template<>
using remote_result_type = typename type::remote_result_type
```

namespace hpx

namespace traits

Variables

```cpp
template<typename T>
constexpr bool is_client_v = is_client<T>::value
```

```cpp
template<typename T>
constexpr bool is_client_or_client_array_v = is_client_or_client_array<T>::value
```

namespace hpx

namespace traits

```cpp
template<typename Policy, typename Enable = void>
struct num_container_partitions
```
Public Static Functions

```cpp
static std::size_t call (Policy const &policy)
```

agas

The contents of this module can be included with the header hpx/modules/agas.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/agas.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
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, naming::id_type>
    using iterate_types_function_type = hpx::util::function<void (std::string const&, components::component_type)>
    using mutex_type = hpx::lcos::local::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)
    addressing_service (util::runtime_configuration const &ini_)
                 ~addressing_service ()
    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)
        Adjust the size of the local AGAS Address resolution cache.
    state get_status () const
```
void set_status(state new_state)

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)

bool is_bootstrap() const

bool is_console() const
   Returns whether this addressing_service represents the console locality.

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)

void register_server_instances()

void garbage_collect_non_blocking(error_code &ec = throws)

void garbage_collect(error_code &ec = throws)

std::int64_t synchronize_with_async_incref(hpx::future<std::int64_t> fut, naming::id_type const &id, std::int64_t compensated_credit)

server::primary_namespace &get_local_primary_namespace_service()

naming::address::address_type get_primary_ns_lva() const

naming::address::address_type get_symbol_ns_lva() const

server::component_namespace *get_local_component_namespace_service()

server::locality_namespace *get_local_locality_namespace_service()

server::symbol_namespace &get_local_symbol_namespace_service()

std::uint64_t get_cache_entries(bool)

std::uint64_t get_cache_hits(bool)

std::uint64_t get_cache_misses(bool)

std::uint64_t get_cache_insertions(bool)

std::uint64_t get_cache_evictions(bool reset)

std::uint64_t get_cache_get_entry_count(bool reset)

std::uint64_t get_cache_insertion_entry_count(bool reset)

std::uint64_t get_cache_update_entry_count(bool reset)

std::uint64_t get_cache_erase_entry_count(bool reset)

std::uint64_t get_cache_get_entry_time(bool reset)

std::uint64_t get_cache_insertion_entry_time(bool reset)
std::uint64_t get_cache_update_entry_time (bool reset)

std::uint64_t get_cache_erase_entry_time (bool reset)

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.

Return 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.

Return This function returns true if a console locality_id exists and returns false 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
• locality_id: [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.

bool get_localities (std::vector<naming::gid_type> &locality_ids, components::component_type type, error_code &ec = throws)
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_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.
bool get_localities (std::vector<naming::gid_type> &locality_ids, error_code &ec = throws)

lcos::future<std::uint32_t> get_num_localities_async (components::component_type type = components::component_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_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

std::uint32_t get_num_localities (error_code &ec = throws) const

lcos::future<std::uint32_t> get_num_overall_threads_async () const

std::uint32_t get_num_overall_threads (error_code &ec = throws) const

lcos::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)

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.

**Return** The function returns the currently associated component type. 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 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.

void iterate_types (iterate_types_function_type const &f, error_code &ec = throws)

std::string get_component_type_name (components::component_type id, error_code &ec = throws)

components::component_type register_factory (naming::gid_type const &locality_id, std::string const &name, error_code &ec = throws)

Register a factory for a specific component type.
This function allows to register a component factory for a given locality and component type.

**Return** 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.

**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.

```cpp
components::component_type register_factory(std::uint32_t locality_id, std::string const &name, error_code &ec = throws)
```

```cpp
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.

**Return** 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.

**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.

```cpp
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.
**Return** This function returns `true`, if this global id got associated with an local address. It returns `false` 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`.

**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.

```cpp
hpx::future<bool> bind_async(naming::gid_type const &id, naming::address const &addr, std::uint32_t locality_id)
```

```cpp
hpx::future<bool> bind_async(naming::gid_type const &id, naming::address const &addr, naming::gid_type const &locality)
```

```cpp
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`).

**Return** This function returns `true`, if the given range was successfully bound. It returns `false` 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`.

**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.

```cpp
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)
```

```cpp
hpx::future<bool> bind_range_async(naming::gid_type const &lower_id, std::uint64_t count, naming::address const &baseaddr, std::uint32_t locality_id)
```

```cpp
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.
**Return** 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.

**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! TODO: confirm that this happens.

**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.

```cpp
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.

**Return** 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.

**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! TODO: confirm that this happens.

**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.

```cpp
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.

**Return** 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.

**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! TODO: confirm that this happens.

**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.

```cpp
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.

**Return**
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.

**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.

```cpp
hpx::future<naming::address> unbind_range_async(naming::gid_type const &lower_id, std::uint64_t count = 1)
```

```cpp
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.

**Return**
This function returns `true` if the passed address refers to an object which lives 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**
- **addr**: [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.

```cpp
bool is_local_address_cached(naming::gid_type const &id, naming::address &addr, error_code &ec = throws)
```

```cpp
bool is_local_lva_encoded_address(std::uint64_t msb)
```

```cpp
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).

**Return** 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.

**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.

```cpp
bool resolve_local ( naming::id_type const &id, naming::address &addr, error_code &ec = throws )
```

```cpp
hpx::future<naming::address> resolve_async ( naming::gid_type const &id )
```

```cpp
hpx::future<naming::address> resolve_async ( naming::id_type const &id )
```

```cpp
hpx::future<naming::id_type> get_colocation_id_async ( naming::id_type const &id )
```

```cpp
bool resolve_full_local ( naming::gid_type const &id, naming::address &addr, error_code &ec = throws )
```

```cpp
hpx::future<naming::address> resolve_full_async ( naming::gid_type const &id )
```

```cpp
hpx::future<naming::address> resolve_full_async ( naming::id_type const &id )
```

```cpp
bool resolve_cached ( naming::gid_type const &id, naming::address &addr, error_code &ec = throws )
```

```cpp
bool resolve_cached ( naming::id_type const &id, naming::address &addr, error_code &ec = throws )
```

```cpp
bool resolve_local ( naming::gid_type const *gids, naming::address *addrs, std::size_t size, boost::dynamic_bitset<> &locals, error_code &ec = throws )
```

```cpp
bool resolve_full_local ( naming::gid_type const *gids, naming::address *addrs, std::size_t size, boost::dynamic_bitset<> &locals, error_code &ec = throws )
```

```cpp
bool resolve_cached ( naming::gid_type const *gids, naming::address *addrs, std::size_t size, boost::dynamic_bitset<> &locals, error_code &ec = throws )
```
lcos::future<
std::int64_t>
incref_async(naming::gid_type const &gid,
std::int64_t credits = 1,

naming::id_type const &keep_alive = naming::invalid_id)

Increment the global reference count for the given id.

Return Whether the operation was successful.
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 incremented.
• 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.

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.

Return The global reference count after the decrement.
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.

hpx::future<
iterate_names_return_type>
iterate_ids(std::string const &pattern)

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 (poosibly using wildcards) to match all existing entries against

bool register_name(std::string const &name,
naming::gid_type const &id,
error_code &ec = throws)

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).

Return The function returns true if the global name was registered. It returns false if the global
name is not registered.
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.
• \texttt{ec}: [in,out] this represents the error status on exit, if this is pre-initialized to \texttt{hpx:throws} the function will throw on error instead.

\begin{verbatim}
lcos::future<bool> register_name_async (std::string const &name, naming::id_type const &id)
bool register_name (std::string const &name, naming::id_type const &id, error_code &ec = throws)
lcos::future<naming::id_type> unregister_name_async (std::string const &name)
Unregister a global name (release any existing association)
This function releases any existing association of the given global name with a global address (id).

Return The function returns \texttt{true} if an association of this global name has been released, and it returns \texttt{false}, if no association existed. Any error results in an exception thrown from this function.

Note As long as \texttt{ec} is not pre-initialized to \texttt{hpx:throws} this function doesn’t throw but returns the result code using the parameter \texttt{ec}. Otherwise it throws an instance of \texttt{hpx::exception}.

Parameters
• \texttt{name}: [in] The global name (string) for which any association with a global address (id) has to be released.
• \texttt{ec}: [in,out] this represents the error status on exit, if this is pre-initialized to \texttt{hpx:throws} the function will throw on error instead.
\end{verbatim}

\begin{verbatim}
naming::id_type unregister_name (std::string const &name, error_code &ec = throws)
\end{verbatim}

\begin{verbatim}
lcos::future<naming::id_type> resolve_name_async (std::string const &name)
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.

Return [out] The id currently associated with the given global name (valid only if the return value is true).

Parameters
• \texttt{name}: [in] The global name (string) for which the currently associated global address has to be retrieved.
• \texttt{ec}: [in,out] this represents the error status on exit, if this is pre-initialized to \texttt{hpx:throws} the function will throw on error instead.

Note As long as \texttt{ec} is not pre-initialized to \texttt{hpx:throws} this function doesn’t throw but returns the result code using the parameter \texttt{ec}. Otherwise it throws an instance of \texttt{hpx::exception}.

\begin{verbatim}
naming::id_type resolve_name (std::string const &name, error_code &ec = throws)
\end{verbatim}

\begin{verbatim}
future<hpx::id_type> on_symbol_namespace_event (std::string const &name, bool call_for_past_events = false)
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.
\end{verbatim}
Return A future instance encapsulating the global id which is causing the registered listener to be 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_forPast_events: [in, optional] Trigger the listener even if the given event has already happened in the past. The default for this parameter is false.

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.

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 &id_base, error_code &ec = throws)

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)

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)

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<naming::id_type, naming::address>> begin_migration (naming::id_type const &id)

start/stop migration of an object

Return Current locality and address of the object to migrate

bool end_migration (naming::id_type const &id)

std::pair<bool, components::pinned_ptr> was_object_migrated (naming::gid_type const &gid, util::unique_function_nonser<components::pinned_ptr>& f) &&

> & &f Maintain list of migrated objects.

hpx::future<void> mark_as_migrated (naming::gid_type const &gid, util::unique_function_nonser<std::pair<bool, hpx::future<void>>> f) &&

> & &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)

Remove the given object from the table of migrated objects.
void `pre_cache_endpoints` (std::vector<parcelset::endpoints_type> const &)

**Public Members**

- `mutex_type gva_cache_mtx_`
- `std::shared_ptr<gva_cache_type> gva_cache_`
- `mutex_type migrated_objects_mtx_`
- `migrated_objects_table_type migrated_objects_table_`
- `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_`
- `mutex_type resolved localities mtx_`
- `resolved localities_type resolved localities_`

**Protected Functions**

- `void launch_bootstrap` (parcelset::endpoints_type const &endpoints, util::runtime_configuration &rtcfg)
- `naming::address resolve_full_postproc` (naming::gid_type const &id, future<primary_namespace::resolved_type> f)
- `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.

namespace hpx

namespace naming

Typedefs

using resolver_client = agas::addressing_service

Functions

agas::addressing_service &get_agas_client ()
agas::addressing_service *get_agas_client_ptr ()

namespace hpx

namespace agas

Functions

bool router_is (state st)

agas_base

The contents of this module can be included with the header hpx/modules/agas_base.hpp. These headers
may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these
at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the
module header hpx/modules/agas_base.hpp, not the particular header in which the functionality you would
like to use is defined. See Public API for a list of names that are part of the public HPX API.
namespace hpx

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.

```plaintext
|-----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 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_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.
```

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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

```
00000000xxxxxxxxxxxxxxxxxxxxxxxxx
Historically unused address space reserved for future use.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxx
Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxx
Prefix of the bootstrap AGAS locality.
00000010000001000000000000001
Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000100000000000002
Address of the component_namespace component on the bootstrap AGAS locality.
00000001000000100000000000003
Address of the symbol_namespace component on the bootstrap AGAS locality.
00000001000000100000000000004
```

(continues on next page)
namespace agas

Typedefs

using iterate_types_function_type = hpx::util::function<void (std::string const &, components::component_type) >

Variables

constexpr char const * const service_name = "/0/agas/
constexpr const std::uint64_t bootstrap_prefix = 0ULL
constexpr const std::uint64_t primary_ns_msb = 0x100000001ULL
constexpr const std::uint64_t primary_ns_lsb = 0x000000001ULL
constexpr const std::uint64_t component_ns_msb = 0x100000001ULL
constexpr const std::uint64_t component_ns_lsb = 0x000000002ULL
constexpr const std::uint64_t symbol_ns_msb = 0x100000001ULL
constexpr const std::uint64_t symbol_ns_lsb = 0x000000003ULL
constexpr const std::uint64_t locality_ns_msb = 0x100000001ULL
constexpr const std::uint64_t locality_ns_lsb = 0x000000004ULL

namespace components

Typedefs

using component_type = std::int32_t

namespace hpx

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 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_runtime_support` the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

**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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

![Reserved Address Ranges]

```cpp
namespace agas
{
    struct component_namespace
    {
    ...
    }
}
```
Public Functions

```cpp
virtual ~component_namespace() = 0
virtual naming::address::address_type ptr() const = 0
virtual naming::address addr() const = 0
virtual naming::id_type gid() const = 0
virtual components::component_type bind_prefix(std::string const &key,
                                             std::uint32_t prefix) = 0
virtual components::component_type bind_name(std::string const &name) = 0
virtual std::vector<std::uint32_t> resolve_id(components::component_type key) = 0
virtual bool unbind(std::string const &key) = 0
virtual void iterate_types(iterate_types_function_type const &f) = 0
virtual std::string get_component_type_name(components::component_type type) = 0
virtual lcos::future<std::uint32_t> get_num_localities(components::component_type type) = 0
virtual void register_server_instance(std::uint32_t)
virtual void unregister_server_instance(error_code&) = 0

server::component_namespace *get_service() = 0
```

namespace hpx

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 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 hpx#components#component_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

**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.

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.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxx
  Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxx
  Prefix of the bootstrap AGAS locality.
00000001000000000000000000000001
  Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000002
  Address of the component_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000003
  Address of the symbol_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000004
  Address of the locality_namespace component on the bootstrap AGAS locality.
```

```namespace agas

**Functions**

```
template<typename Char, typename Traits>
std::basic_ostream<Char, Traits>& operator<<(std::basic_ostream<Char, Traits>& os, gva const &addr)

struct gva
```
Public Types

```cpp
using component_type = std::int32_t
using lva_type = std::uint64_t
```

Public Functions

```cpp
gva ()
gva (naming::gid_type const &p, component_type t = components::component_invalid, std::uint64_t c = 1, lva_type a = 0, std::uint64_t o = 0)
gva (naming::gid_type const &p, component_type t, std::uint64_t c, void *a, std::uint64_t o = 0)
gva (lva_type a)
gva (void *a)
gva &operator= (lva_type a)
gva &operator= (void *a)
bool operator== (gva const &rhs) const
bool operator!= (gva const &rhs) const
void lva (lva_type a)
void lva (void *a)
lva_type lva () const
lva_type lva (naming::gid_type const &gid, naming::gid_type const &gidbase) const
gva resolve (naming::gid_type const &gid, naming::gid_type const &gidbase) const
```

Public Members

```cpp
naming::gid_type prefix
component_type type
std::uint64_t count
std::uint64_t offset
```

Private Functions

```cpp
template<class Archive>
void save (Archive &ar, const unsigned int) const

template<class Archive>
void load (Archive &ar, const unsigned int version)
```
Private Members

*lva_type* `lva_`

Friends

friend hpx::agas::hpx::serialization::access

namespace hpx

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).
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_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.
```

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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

```
00000000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  Historically unused address space reserved for future use.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxxxx
  Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxxx
  Prefix of the bootstrap AGAS locality.
00000001000000010000000000000001
```

(continues on next page)
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.

namespace agas

struct locality_namespace

Public Functions

virtual ~locality_namespace ()
virtual naming::address::address_type ptr () const = 0
virtual naming::address addr () const = 0
virtual naming::id_type gid () const = 0
virtual std::uint32_t allocate (parcelset::endpoints_type const &endpoints,
std::uint64_t count, std::uint32_t num_threads, naming::gid_type const &suggested_prefix) = 0
virtual void free (naming::gid_type const &locality) = 0
virtual std::vector<std::uint32_t> localities () = 0
virtual parcelset::endpoints_type resolve_locality (naming::gid_type const &locality) = 0
virtual std::uint32_t get_num_localities () = 0
virtual hpx::future<std::uint32_t> get_num_localities_async () = 0
virtual std::vector<std::uint32_t> get_num_threads () = 0
virtual hpx::future<std::vector<std::uint32_t>> get_num_threads_async () = 0
virtual std::uint32_t get_num_overall_threads () = 0
virtual hpx::future<std::uint32_t> get_num_overall_threads_async () = 0
virtual void register_server_instance (std::uint32_t)
virtual void unregister_server_instance (error_code&)
virtual server::locality_namespace *get_service ()
namespace hpx
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 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_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

00000000xxxxxxxxxxxxxxxxxxxxxxxxxxxx
   Historically unused address space reserved for future use.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxxxx
   Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxxxxxx
   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.
00000000100000001000000000000004

(continues on next page)
namespace agas

struct primary_namespace

Public Types

typedef hpx::tuple<naming::gid_type, gva, naming::gid_type> resolved_type

Public Functions

primary_namespace ()
~primary_namespace ()
naming::address::address_type ptr () const
naming::address addr () const
naming::id_type gid () const

hpx::future<std::pair<naming::id_type, naming::address>> begin_migration (naming::gid_type const &id)

bool end_migration (naming::gid_type const &id)

bool bind_gid (gva const &g, naming::gid_type const &id, naming::gid_type const &locality)

future<bool> bind_gid_async (gva g, naming::gid_type id, naming::gid_type locality)

resolved_type resolve_gid (naming::gid_type const &id)

future<resolved_type> resolve_full (naming::gid_type id)

future<id_type> colocate (naming::gid_type id)

naming::address unbind_gid (std::uint64_t count, naming::gid_type const &id)

future<naming::address> unbind_gid_async (std::uint64_t count, naming::gid_type const &id)

future<std::int64_t> increment_credit (std::int64_t credits, naming::gid_type lower, naming::gid_type upper)

std::pair<naming::gid_type, naming::gid_type> allocate (std::uint64_t count)

void set_local_locality (naming::gid_type const &g)

void register_server_instance (std::uint32_t locality_id)

void unregister_server_instance (error_code &ec)

server::primary_namespace &get_service ()
Public Static Functions

```cpp
static naming::gid_type get_service_instance (std::uint32_t service_locality_id)
static naming::gid_type get_service_instance (naming::gid_type const &dest, error_code &ec = throws)
static naming::gid_type get_service_instance (naming::id_type const &dest)
static bool is_service_instance (naming::gid_type const &gid)
static bool is_service_instance (naming::id_type const &id)
```

Private Members

```cpp
std::unique_ptr<server::primary_namespace> server_
```

namespace hpx

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).
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 \hpx\#components#component_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.
```

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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.
Historically unused address space reserved for future use.
Address space for LVA-encoded GIDs.
Prefix of the bootstrap AGAS locality.
Address of the primary_namespace component on the bootstrap AGAS locality.
Address of the component_namespace component on the bootstrap AGAS locality.
Address of the symbol_namespace component on the bootstrap AGAS locality.
Address of the locality_namespace component on the bootstrap AGAS locality.

namespace agas

struct symbol_namespace

Public Types

using server_type = server::symbol_namespace
using iterate_names_return_type = std::map<std::string, naming::id_type>

Public Functions

symbol_namespace()
~symbol_namespace()
naming::address_type ptr() const
naming::address addr() const
naming::id_type gid() const
hpx::future<bool> bind_async(std::string key, naming::gid_type gid)
bool bind(std::string key, naming::gid_type gid)

hpx::future<naming::id_type> resolve_async(std::string key) const
naming::id_type resolve(std::string key) const

hpx::future<naming::id_type> unbind_async(std::string key)

naming::id_type unbind(std::string key)

hpx::future<bool> on_event(std::string const &name, bool call_for_past_events,
                           hpx::id_type lco)
```cpp
hpx::future<iterate_names_return_type> iterate_async(std::string const &pattern) const
iterate_names_return_type iterate(std::string const &pattern) const
void register_server_instance(std::uint32_t locality_id)
void unregister_server_instance(error_code &ec)

server::symbol_namespace &get_service()
```

**Public Static Functions**

```cpp
static naming::gid_type get_service_instance(std::uint32_t service_locality_id)
static naming::gid_type get_service_instance(naming::gid_type const &dest, error_code &ec = throws)
static naming::gid_type get_service_instance(naming::id_type const &dest)
static bool is_service_instance(naming::gid_type const &gid)
static bool is_service_instance(naming::id_type const &id)
static naming::id_type symbol_namespace_locality(std::string const &key)
```

**Private Members**

```cpp
std::unique_ptr<server_type> server_
```

**namespace hpx**

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.

```plaintext
|-----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 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
```

(continues on next page)
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_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

**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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter `x` represents a single-byte wild card.

```
00000000xxxxxxxxxxxxxxxxxxxxxxxxxx
   Historically unused address space reserved for future use.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxxxxx
   Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxxxx
   Prefix of the bootstrap AGAS locality.
00000010000000100000000000000001
   Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000100000000000000002
   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.
```

```namespace agas

**Functions**

`naming::gid_type bootstrap_component_namespace_gid()`

`naming::id_type bootstrap_component_namespace_id()`

**Variables**

`constexpr char const *const component_namespace_service_name = "component/"

`struct component_namespace : public components::fixed_component_base<component_namespace>`
Public Types

using mutex_type = lcos::local::spinlock
using base_type = components::fixed_component_base<component_namespace>
using component_id_type = components::component_type
using prefixes_type = std::set<std::uint32_t>
using component_id_table_type = std::unordered_map<std::string, component_id_type>
using factory_table_type = std::map<component_id_type, prefixes_type>

Public Functions

component_namespace()
void finalize()
void register_server_instance (char const *servicename, error_code &ec = throws)
void unregister_server_instance (error_code &ec = throws)
components::component_type bind_prefix (std::string const &key, std::uint32_t prefix)
components::component_type bind_name (std::string const &name)
std::vector<std::uint32_t> resolve_id (components::component_type key)
bool unbind (std::string const &key)
void iterate_types (iterate_types_function_type const &f)
std::string get_component_type_name (components::component_type type)
std::uint32_t get_num_localities (components::component_type type)

HPX_DEFINE_COMPONENT_ACTION (component_namespace, bind_prefix)
HPX_DEFINE_COMPONENT_ACTION (component_namespace, bind_name)
HPX_DEFINE_COMPONENT_ACTION (component_namespace, resolve_id)
HPX_DEFINE_COMPONENT_ACTION (component_namespace, unbind)
HPX_DEFINE_COMPONENT_ACTION (component_namespace, iterate_types)
HPX_DEFINE_COMPONENT_ACTION (component_namespace, get_component_type_name)
HPX_DEFINE_COMPONENT_ACTION (component_namespace, get_num_localities)
Public Members

`counter_data` **counter_data**

Public Static Functions

```cpp
static void register_counter_types (error_code &ec = throws)
static void register_global_counter_types (error_code &ec = throws)
```

Private Members

```cpp
mutex_type mutex_
component_id_table_type component_ids_
factory_table_type factories_
component_id_type type_counter
std::string instance_name_
```

struct **counter_data**

Public Types

```cpp
typedef lcos::local::spinlock mutex_type
```

Public Functions

```cpp
HPX_NON_COPYABLE (counter_data)
counter_data ()
std::int64_t get_bind_prefix_count (bool)
std::int64_t get_bind_name_count (bool)
std::int64_t get_resolve_id_count (bool)
std::int64_t get_unbind_name_count (bool)
std::int64_t get_iterate_types_count (bool)
std::int64_t get_component_type_name_count (bool)
std::int64_t get_num_localities_count (bool)
std::int64_t get_overall_count (bool)
std::int64_t get_bind_prefix_time (bool)
std::int64_t get_bind_name_time (bool)
std::int64_t get_resolve_id_time (bool)
std::int64_t get_unbind_name_time (bool)
```
std::int64_t get_iterate_types_time(bool)
std::int64_t get_component_type_name_time(bool)
std::int64_t get_num_localities_time(bool)
std::int64_t get_overall_time(bool)
void increment_bind_prefix_count()
void increment_bind_name_count()
void increment_resolve_id_count()
void increment_unbind_name_count()
void increment_iterate_types_count()
void increment_get_component_type_name_count()
void increment_num_localities_count()
void enable_all()

Public Members

Public Functions

api_counter_data bind_prefix_
api_counter_data bind_name_
api_counter_data resolve_id_
api_counter_data unbind_name_
api_counter_data iterate_types_
api_counter_data get_component_type_name_
api_counter_data num_localities_

struct api_counter_data

Public Members

api_counter_data()
HPX Documentation, master

------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).
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_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

00000000xxxxxxxxxxxxxxxxxxxxxxxxxx
   Historically unused address space reserved for future use.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxxxx
   Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxxxx
   Prefix of the bootstrap AGAS locality.
00000010000000000000000000000001
   Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000002
   Address of the component_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000003
   Address of the symbol_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000004
   Address of the locality_namespace component on the bootstrap AGAS locality.

namespace agas

2.8. API reference 1311
Functions

naming::gid_type bootstrap_locality_namespace_gid()
naming::id_type bootstrap_locality_namespace_id()

namespace server

Variables

constexpr char const *const locality_namespace_service_name = "locality/

struct locality_namespace : public components::fixed_component_base<locality_namespace>

Public Types

using mutex_type = lcos::local::spinlock
using base_type = components::fixed_component_base<locality_namespace>
using component_type = std::int32_t
using partition_type = hpx::tuple<parcelset::endpoints_type, std::uint32_t>
using partition_table_type = std::map<std::uint32_t, partition_type>

Public Functions

locality_namespace (primary_namespace *primary)
void finalize ()
void register_server_instance (char const *servicename, error_code &ec = throws)
void unregister_server_instance (error_code &ec = throws)
std::uint32_t allocate (parcelset::endpoints_type const &endpoints, std::uint64_t count,
                      std::uint32_t num_threads, naming::gid_type suggested_prefix)
parcelset::endpoints_type resolve_locality (naming::gid_type const &locality)
void free (naming::gid_type const &locality)
std::vector<std::uint32_t> localities ()
std::uint32_t get_num_localities ()
std::vector<std::uint32_t> get_num_threads ()
std::uint32_t get_num_overall_threads ()
HPX_DEFINE_COMPONENT_ACTION (locality_namespace, allocate)
HPX_DEFINE_COMPONENT_ACTION (locality_namespace, free)
HPX_DEFINE_COMPONENT_ACTION (locality_namespace, localities)
HPX_DEFINE_COMPONENT_ACTION (locality_namespace, resolve_locality)
HPX_DEFINE_COMPONENT_ACTION (locality_namespace, get_num_localities)
HPX_DEFINE_COMPONENT_ACTION (locality_namespace, get_num_threads)
HPX_DEFINE_COMPONENT_ACTION (locality_namespace, get_num_overall_threads)

Public Members

counter_data counter_data_

Private Members

mutex_type mutex_
std::string instance_name_
partition_table_type partitions_
std::uint32_t prefix_counter_
primary_namespace *primary_

struct counter_data

Public Types

typedef lcos::local::spinlock mutex_type

Public Functions

HPX_NON_COPYABLE (counter_data)

counter_data ()
std::int64_t get_allocate_count (bool)
std::int64_t get_resolve_locality_count (bool)
std::int64_t get_free_count (bool)
std::int64_t get_localities_count (bool)
std::int64_t get_num_localities_count (bool)
std::int64_t get_num_threads_count (bool)
std::int64_t get_resolved_localities_count (bool)
std::int64_t get_overall_count (bool)
std::int64_t get_allocate_time (bool)
std::int64_t get_resolve_locality_time (bool)
std::int64_t get_free_time (bool)
std::int64_t get_localities_time(bool)
std::int64_t get_num_localities_time(bool)
std::int64_t get_num_threads_time(bool)
std::int64_t get_resolved_localities_time(bool)
std::int64_t get_overall_time(bool)
void increment_allocate_count()
void increment_resolve_locality_count()
void increment_free_count()
void increment_localities_count()
void increment_num_localities_count()
void increment_num_threads_count()
void enable_all()

Public Members

api_counter_data allocate_
api_counter_data resolve_locality_
api_counter_data free_
api_counter_data localities_
api_counter_data num_localities_
api_counter_data num_threads_

struct api_counter_data

Public Functions

api_counter_data()

Public Members

std::atomic< std::int64_t > count_
std::atomic< std::int64_t > time_
bool enabled_
Variables

**HPX_ACTION_USES_MEDIUM_STACK** (hpx::agas::server::primary_namespace::allocate_action)

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).
- **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 hpx#components#component_runtime_support the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

**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.

The following address ranges are reserved. Some are either explicitly or implicitly protected by AGAS. The letter x represents a single-byte wild card.

```
00000000xxxxxxxxxxxxxxxxxxxxxxxxxxx
  Historically unused address space reserved for future use.
xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxxx
  Address space for LVA-encoded GIDs.
00000001xxxxxxxxxxxxxxxxxxxxxxxxxxx
  Prefix of the bootstrap AGAS locality.
00000001000000000000000000000001
  Address of the primary_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000002
  Address of the component_namespace component on the bootstrap AGAS locality.
```
namespace agas

Functions

naming::gid_type bootstrap_primary_namespace_gid()
naming::id_type bootstrap_primary_namespace_id()

namespace server

Variables

cconstexpr char const *const primary_namespace_service_name = "primary/"

struct primary_namespace : public components::fixed_component_base<primary_namespace>

Public Types

using mutex_type = lcos::local::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

primary_namespace()  
void finalize()  
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)
bool bind_gid(gva const &g, naming::gid_type id, naming::gid_type const &locality)
std::pair<naming::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)
naming::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)

HPX_DEFINE_COMPONENT_ACTION (primary_namespace, allocate)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, bind_gid)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, colocate)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, begin_migration)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, end_migration)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, decrement_credit)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, increment_credit)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, resolve_gid)
HPX_DEFINE_COMPONENT_ACTION (primary_namespace, unbind_gid)

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

void wait_for_migration_locked (std::unique_lock<mutex_type> &l, naming::gid_type const &id, error_code &ec)

resolved_type resolve_gid_locked (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 &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_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 &free_list, naming::gid_type const &lower, naming::gid_type const &upper, error_code &ec)

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 ()

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)
```cpp
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

api_counter_data()

Public Members

std::atomic<std::int64_t> count_
std::atomic<std::int64_t> time_
bool enabled_

struct free_entry

Public Functions

free_entry (agas::gva gva, naming::gid_type const &gid, naming::gid_type const &loc)

Public Members

agas::gva gva_
naming::gid_type gid_
naming::gid_type locality_

namespace hpx

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 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 `hpx\#components\#component_runtime_support` the high 24 bits are zeroed and the low 64 bits hold the LVA of the component.

**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.

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.
00000100000000000000000000000001
  Address of the primary_namespace component on the bootstrap AGAS locality.
00000100000000000000000000000002
  Address of the component_namespace component on the bootstrap AGAS locality.
00000010000000000000000000000003
  Address of the symbol_namespace component on the bootstrap AGAS locality.
00000001000000000000000000000004
  Address of the locality_namespace component on the bootstrap AGAS locality.
```

nenamespace agas

**Functions**

`naming::gid_type bootstrap_symbol_namespace_gid()`

`naming::id_type bootstrap_symbol_namespace_id()`

**namespace server**

**Variables**

```cpp
constexpr char const *const symbol_namespace_service_name = "symbol/"

struct symbol_namespace : public components::fixed_component_base<symbol_namespace>
```
Public Types

using mutex_type = lcos::local::spinlock
using base_type = components::fixed_component_base<symbol_namespace>
using iterate_names_return_type = std::map<std::string, naming::gid_type>
using gid_table_type = std::map<std::string, std::shared_ptr<naming::gid_type>>
using on_event_data_map_type = std::multimap<std::string, hpx::id_type>

Public Functions

symbol_namespace()  
void finalize()  

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)

bool bind (std::string key, naming::gid_type gid)

naming::gid_type resolve (std::string const &key)

naming::gid_type unbind (std::string const &key)

iterate_names_return_type iterate (std::string const &pattern)

bool on_event (std::string const &name, bool call_for_past_events, hpx::id_type lco)

HPX_DEFINE_COMPONENT_ACTION (symbol_namespace, bind)

HPX_DEFINE_COMPONENT_ACTION (symbol_namespace, resolve)

HPX_DEFINE_COMPONENT_ACTION (symbol_namespace, unbind)

HPX_DEFINE_COMPONENT_ACTION (symbol_namespace, iterate)

HPX_DEFINE_COMPONENT_ACTION (symbol_namespace, on_event)

Public Members

counter_data counter_data_

Public Static Functions

static void register_counter_types (error_code &ec = throws)

static void register_global_counter_types (error_code &ec = throws)
Private Members

mutex_type mutex_
gid_table_type gids_
std::string instance_name_
on_event_data_map_type on_event_data_

struct counter_data

Public Types

typedef lcos::local::spinlock mutex_type

Public Functions

HPX_NON_COPYABLE (counter_data)
counter_data ()
std::int64_t get_bind_count (bool)
std::int64_t get_resolve_count (bool)
std::int64_t get_unbind_count (bool)
std::int64_t get_iterate_names_count (bool)
std::int64_t get_on_event_count (bool)
std::int64_t get_overall_count (bool)
std::int64_t get_bind_time (bool)
std::int64_t get_resolve_time (bool)
std::int64_t get_unbind_time (bool)
std::int64_t get_iterate_names_time (bool)
std::int64_t get_on_event_time (bool)
std::int64_t get_overall_time (bool)
void increment_bind_count ()
void increment_resolve_count ()
void increment_unbind_count ()
void increment_iterate_names_count ()
void increment_on_event_count ()
void enable_all ()
Public Members

*api_counter_data bind_*
*api_counter_data resolve_*
*api_counter_data unbind_*
*api_counter_data iterate_names_*
*api_counter_data on_event_*

**struct api_counter_data**

Public Functions

*api_counter_data()*

Public Members

*std::atomic<std::int64_t> count_*
*std::atomic<std::int64_t> time_*
*bool enabled_*

async_colocated

The contents of this module can be included with the header `hpx/modules/async_colocated.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we **strongly** suggest only including the module header `hpx/modules/async_colocated.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

**Defines**

**HPX_REGISTER_ASYNC_COLOCATED_DECLARATION** *(Action, Name)*

**HPX_REGISTER_ASYNC_COLOCATED** *(Action, Name)*

**namespace hpx**

**Functions**

*naming::id_type get_colocation_id(launch::sync_policy, naming::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.

**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`. 

---

1324 Chapter 2. What’s so special about HPX?
See `hpx::get_colocation_id()`

**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.

```cpp
lcos::future<naming::id_type> get_colocation_id(naming::id_type const &id)
```

Asynchronously return the id of the locality where the object referenced by the given id is currently located on.

See `hpx::get_colocation_id(launch::sync_policy)`

**Parameters**

- **id**: [in] The id of the object to locate.

**Defines**

`HPX_REGISTER_APPLY_COLOCATED_DECLARATION (Action, Name)`

`HPX_REGISTER_APPLY_COLOCATED (action, name)`

```cpp
namespace hpx
```

```cpp
namespace util
```

```cpp
namespace functional
```

**Functions**

```cpp
template<typename Bound>
functional::detail::apply_continuation_impl<Bound, hpx::util::unused_type> apply_continuation (Bound &&&bound)
```

```cpp
template<typename Bound, typename Continuation>
functional::detail::apply_continuation_impl<Bound, Continuation> apply_continuation (Bound &&&bound, Continuation &&c)
```

```cpp
template<typename Bound>
functional::detail::async_continuation_impl<Bound, hpx::util::unused_type> async_continuation (Bound &&&bound)
```

```cpp
template<typename Bound, typename Continuation>
```
functional::detail::async_continuation_impl<Bound, Continuation> \texttt{async\_continuation}(Bound &&\texttt{bound}, Continuation &&\texttt{c})

\textbf{struct extract\_locality}

\textbf{Public Functions}

\texttt{naming::id\_type operator() (naming::id\_type const &locality\_id, naming::id\_type const &id) const}

\textbf{namespace hpx}

\textbf{namespace components}

\textbf{namespace server}

\textbf{Functions}

\texttt{void destroy\_component (naming::gid\_type const &gid, naming::address const &addr)}

\texttt{template<\texttt{typename Component}> void destroy (naming::gid\_type const &gid, naming::address const &addr)}

\texttt{template<\texttt{typename Component}> void destroy (naming::gid\_type const &gid)}

\textbf{async\_distributed}

The contents of this module can be included with the header \texttt{hpx/modules/async\_distributed.hpp}. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header \texttt{hpx/modules/async\_distributed.hpp}, not the particular header in which the functionality you would like to use is defined. See \textit{Public API} for a list of names that are part of the public HPX API.

\textbf{namespace hpx}
Functions

template<typename Action, typename F, typename ...Ts>
auto async (F &&f, Ts&&... ts)

namespace hpx

Functions

template<typename Action, typename F, typename ...Ts>
auto async_cb (F &&f, Ts&&... ts)

namespace hpx

Functions

template<typename Action, typename F, typename ...Ts>
lcos::future<
typename traits::promise_local_result<
typename
detail::result_of_async_continue<Action, Cont>::type>::type>
async_continue (Cont &&cont, naming::id_type const &gid, Ts&&... vs)

template<typename Component, typename Signature, typename Derived, typename Cont, typename ...Ts>
lcos::future<
typename traits::promise_local_result<
typename
detail::result_of_async_continue<Derived, Cont>::type>::type>
async_continue (hpx::actions::basic_action<Component, Signature, Derived>, Cont &&cont, naming::id_type const &gid, Ts&&... vs)

template<typename Action, typename Cont, typename DistPolicy, typename ...Ts>


std::enable_if<traits::is_distribution_policy<DistPolicy>::value, lcos::future<
    typename traits::promise_local_result<typename
detail::result_of_async_continue<Action, Cont>::type>::type>>::type

namespace hpx

Functions

template<
    typename Component, typename Signature, typename Derived, typename Cont,
    typename DistPolicy, typename ...
>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, lcos::future<
    typename traits::promise_local_result<typename
detail::result_of_async_continue<Derived, Cont>::type>::type>>::type async_continue(
    Cont &&cont,
    DistPolicy const &policy,
    Ts &&... vs)


template<
    typename Component, typename Signature, typename Derived, typename Cont,
    typename Callback, typename ...
>
lcos::future<
    typename traits::promise_local_result<typename
detail::result_of_async_continue<
        hpx::actions::basic_action<Component, Signature, Derived>, Cont>
    >::type>::type async_continue_cb(
    Cont &&cont,
    hpx::id_type const &gid,
    Callback &&cb,
    Ts &&... vs)


lcos::future<
    typename traits::promise_local_result<
        typename detail::result_of_async_continue<
            Derived, Cont>::type>::type>
      async_continue_cb

template<typename Action, typename Cont, typename DistPolicy, typename Callback, typename ... Ts>
    std::enable_if<
        traits::is_distribution_policy<
            DistPolicy>::value, lcos::future<
                typename traits::promise_local_result<
                    typename detail::result_of_async_continue<
                        Action, Cont>::type>::type>>::type
    async_continue_cb

template<typename Component, typename Signature, typename Derived, typename Cont, typename DistPolicy, typename Callback, typename ... Ts>
    std::enable_if<
        traits::is_distribution_policy<
            DistPolicy>::value, lcos::future<
                typename traits::promise_local_result<
                    typename detail::result_of_async_continue<
                        Derived, Cont>::type>::type>>::type
    async_continue_cb

template<>
    struct get_lva<lcos::base_lco>

2.8. API reference
Public Static Functions

```cpp
static lcos::base_lco *call (naming::address_type lva)
```

namespace hpx

```cpp
template<>
struct get_lva<lcos::base_lco const>
```

Public Static Functions

```cpp
static lcos::base_lco const *call (naming::address_type lva)
```

namespace lcos

```cpp
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

```cpp
typedef components::managed_component<base_lco> wrapping_type

typedef base_lco base_type_holder
```
Public Functions

```cpp
virtual void set_event() = 0
virtual void set_exception(const std::exception_ptr& e)
virtual void connect(naming::id_type const&)
virtual void disconnect(naming::id_type const&)
virtual ~base_lco()
    Destructor, needs to be virtual to allow for clean destruction of derived objects
virtual void finalize()
    finalize() will be called just before the instance gets destructed
```

**Parameters**

- `e`: [in] The exception encapsulating the error to report to this LCO instance.

```cpp
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(const std::exception_ptr& 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.

```cpp
void connect_nonvirt(naming::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

```cpp
void disconnect_nonvirt(naming::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**

```cpp
HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco, set_event_nonvirt, set_event_action)
```

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.
HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco, set_exception_nonvirt, set_exception_action)

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.

HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco, connect_nonvirt, connect_action)

The connect_action may be used to.

HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco, disconnect_nonvirt, disconnect_action)

The set_exception_action may be used to.

Public Static Functions

static components::component_type get_component_type ()

static void set_component_type (components::component_type type)

Defines

HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION (...)
HPX_REGISTER_BASE_LCO_WITH_VALUE_DECLARATION_2 (...)
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_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_1 (...)
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 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

template<>
using wrapping_type = typename detail::base_lco_wrapping_type<ComponentTag, base_lco_with_value>::type

template<>
using base_type_holder = base_lco_with_value

Public Functions

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.

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)
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.
HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco_with_value, get_value_nonvirt, get_value_action)

The get_value_action may be used to query the value this LCO instance exposes as its ‘result’ value.

Public Static Functions

static components::component_type get_component_type()
static void set_component_type(components::component_type type)

Protected Types

typedef std::conditional< std::is_void<Result>::value, util::unused_type, Result>::type result_type

Protected Functions

~base_lco_with_value()
Destructor, needs to be virtual to allow for clean destruction of derived objects

void set_event()

void set_event_nonvirt (std::false_type)

void set_event_nonvirt (std::true_type)

virtual void set_value(RemoteResult &&result) = 0

virtual result_type get_value() = 0

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

template<>
using wrapping_type = typename detail::base_lco_wrapping_type<ComponentTag, base_lco_with_value>::type

template<>
using base_type_holder = base_lco_with_value

template<>
using set_value_action = typename base_lco::set_event_action
Public Functions

void get_value()

HPX_DEFINE_COMPONENT_DIRECT_ACTION(base_lco_with_value, get_value, get_value_action)

Protected Functions

~base_lco_with_value()

Destructor, needs to be virtual to allow for clean destruction of derived objects

namespace hpx

namespace serialization

Functions

template<typename Archive, typename F, typename ...Ts>
void serialize(Archive &ar, ::hpx::util::detail::bound_action<F, Ts...> &bound, unsigned int
const version = 0)

namespace util

Functions

template<typename Action, typename ...Ts, typename Enable = typename std::enable_if<std::is_action<typename std::decay<Action>::type>::value>::type>

detail::bound_action<typename std::decay<Action>::type, typename util::make_index_pack<sizeof...(Ts)>::type, typename std::decay<Ts>::type...>


template<typename Component, typename Signature, typename Derived, typename ...Ts>
detail::bound_action<Derived, typename util::make_index_pack<sizeof...(Ts)>::type, typename std::decay<Ts>::type...>


template<>
struct typed_continuation< void, util::unused_type > : public hpx::actions::continuation
Public Types

template<>
using result_type = void

Public Functions

typed_continuation()

typed_continuation (naming::id_type const &id)

typed_continuation (naming::id_type &id)

template<typename F>
typed_continuation (naming::id_type const &id, F &&f)

template<typename F>
typed_continuation (naming::id_type &id, F &&f)

typed_continuation (naming::id_type const &id, naming::address &&addr)

typed_continuation (naming::id_type &id, naming::address &&addr)

template<typename F>
typed_continuation (naming::id_type const &id, naming::address &&addr, F &&f)

template<typename F>
typed_continuation (naming::id_type &id, naming::address &&addr, F &&f)

template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, typed_continuation>::value>::type>
typed_continuation (F &&f)

typed_continuation (typed_continuation&&)

typed_continuation &operator= (typed_continuation&&)

void trigger()

void trigger_value (util::unused_type&&)

void trigger_value (util::unused_type const&)

Private Types

template<>
using function_type = util::unique_function< void (naming::id_type) >
Private Functions

void serialize(hpx::serialization::input_archive &ar, unsigned)
void serialize(hpx::serialization::output_archive &ar, unsigned)

Private Members

function_type f_

Friends

friend hpx::serialization::access
serialization support

namespace hpx

namespace actions

class continuation
Subclassed by hpx::actions::typed_continuation< Result, Result >,
hpx::actions::typed_continuation< void, util::unused_type >

Public Types

typedef void continuation_tag

Public Functions

continuation()
continuation(naming::id_type const &id)
continuation(naming::id_type &&id)
continuation(naming::id_type const &id, naming::address &addr)
continuation(naming::id_type &&id, naming::address &addr)
continuation(continuation &&o)
continuation &operator=(continuation &&o)
void trigger_error(std::exception_ptr const &e)
void trigger_error(std::exception_ptr &e)
void serialize(hpx::serialization::input_archive &ar, unsigned)
void serialize(hpx::serialization::output_archive &ar, unsigned)
constexpr naming::id_type const &get_id() const
constexpr naming::address get_addr() const
Protected Attributes

naming::id_type id_
naming::address addr_

template<typename Result, typename RemoteResult>
struct typed_continuation

Public Functions

typed_continuation()
typed_continuation(naming::id_type const &id)
typed_continuation(naming::id_type &id)
template<typename F>
typed_continuation(naming::id_type const &id, F &&f)
template<typename F>
typed_continuation(naming::id_type &&id, F &&f)
typed_continuation(naming::id_type const &id, naming::address &&addr)
typed_continuation(naming::id_type &&id, naming::address &&addr)
template<typename F>
typed_continuation(naming::id_type const &id, naming::address &&addr, F &&f)
template<typename F>
typed_continuation(naming::id_type &&id, naming::address &&addr, F &&f)

template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename decay<F>::type, typed_continuation>::value>::type>
typed_continuation(F &&f)
typed_continuation(typed_continuation&&)
void trigger_value(RemoteResult &&result)

Private Types

template<>
using base_type = typed_continuation<RemoteResult>
template<>
using function_type = util::unique_function<void (naming::id_type, RemoteResult) >
Private Functions

template<typename Archive>
void serialize (Archive &ar, unsigned)

Friends

friend hpx::actions::hpx::serialization::access
serialization support

Public Types

template<>
using result_type = Result

Public Functions

typed_continuation ()
typed_continuation (naming::id_type const &id)
typed_continuation (naming::id_type &&id)
template<typename F>
typed_continuation (naming::id_type const &id, F &&f)
template<typename F>
typed_continuation (naming::id_type &&id, F &&f)
typed_continuation (naming::id_type const &id, naming::address &&addr)
typed_continuation (naming::id_type &&id, naming::address &&addr)
template<typename F>
typed_continuation (naming::id_type const &id, naming::address &&addr, F &&f)
template<typename F>
typed_continuation (naming::id_type &&id, naming::address &&addr, F &&f)
template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, typed_continuation>::value>::type>
typed_continuation (F &&f)
typed_continuation (typed_continuation &&)
typed_continuation & operator= (typed_continuation &&)
void trigger_value (Result &&result)
Protected Attributes

function_type f_

Private Types

template<>
using function_type = util::unique_function<void (naming::id_type, Result)>

Private Functions

template<typename Archive>
void serialize (Archive & ar, unsigned)

Friends

friend hpx::actions::hpx::serialization::access
serialization support

template<>
struct typed_continuation<void, util::unused_type> : public hpx::actions::continuation

Public Types

template<>
using result_type = void

Public Functions

typed_continuation ()
typed_continuation (naming::id_type const & id)
typed_continuation (naming::id_type & & id)
template<typename F>
typed_continuation (naming::id_type const & id, F & & f)
template<typename F>
typed_continuation (naming::id_type & & id, F & & f)
typed_continuation (naming::id_type const & id, naming::address & & addr)
typed_continuation (naming::id_type & & id, naming::address & & addr)
template<typename F>
typed_continuation (naming::id_type const & id, naming::address & & addr, F & & f)
template<typename F>
typed_continuation (naming::id_type & & id, naming::address & & addr, F & & f)
template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, typed_continuation>::value>::type>
typed_continuation (F & & f)
**Private Types**

```cpp
template<>
using function_type = util::unique_function<void(naming::id_type)>
```

**Private Functions**

```cpp
void serialize(hpx::serialization::input_archive &ar, unsigned)
void serialize(hpx::serialization::output_archive &ar, unsigned)
```

**Private Members**

```cpp
function_type f_
```

**Friends**

```cpp
friend hpx::actions::hpx::serialization::access
serialization support
```

```cpp
namespace hpx
```

```cpp
namespace actions
```

```cpp
template<

namespace hpx
```

```cpp
namespace actions
```

```cpp
struct continuation2_impl
```

**Public Functions**

```cpp
continuation2_impl()
```

```cpp
continuation2_impl(Cont_, &cont, hpx::id_type const &target, F_ &&f)
```

```cpp
virtual ~continuation2_impl()
```

```cpp
template<
```
util::invoke_result<function_type, hpx::id_type, typename util::invoke_result<cont_type, hpx::id_type, T>::type>::type

Private Types

template<>
using cont_type = typename std::decay<Cont>::type

template<
using function_type = typename std::decay<F>::type

Private Functions

template<typename Archive>
void serialize (Archive &ar, unsigned int const)

Private Members

ccont_type cont_

hpx::id_type target_

function_type f_

Friends

friend hpx::actions::hpx::serialization::access

namespace hpx

namespace actions

Functions

template<typename Result, typename RemoteResult, typename F, typename ...Ts>
void trigger (typed_continuation<Result, RemoteResult>&&, F&&, Ts&&...)

namespace hpx

namespace actions

template<typename Cont>
struct continuation_impl
Public Functions

continuation_impl()

template<typename Cont>
continuation_impl(Cont &&cont, hpx::id_type const &target)

virtual ~continuation_impl()

template<typename T>
util::invoke_result<cont_type, hpx::id_type, T>::type operator() (hpx::id_type const &lco, T &&t) const

Private Types

template<>
using cont_type = typename std::decay<Cont>::type

Private Functions

template<typename Archive>
void serialize(Archive &ar, unsigned int const)

Private Members

cont_type cont_

hpx::id_type target_

Friends

friend hpx::actions::hpx::serialization::access

namespace hpx

Functions

template<typename Action, typename T0, typename ...Ts, typename Enable = typename std::enable_if<traits::is_action<Action>::value>::type>
auto dataflow (T0 &&t0, Ts&&... ts)

template<typename Action, typename Allocator, typename T0, typename ...Ts, typename Enable = typename std::enable_if<traits::is_action<Action>::value>::type>
auto dataflow_alloc (Allocator const &alloc, T0 &&t0, Ts&&... ts)

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> 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.

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).

```cpp
// Create the promise (the expected result is a id_type)
lcos::promise<naming::id_type> p;

// Get the associated future
future<naming::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.
naming::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.
Functions

```
hpx::actions::set_lco_value_continuation make_continuation()
```

```
template<typename Cont>
    hpx::actions::continuation_impl<typename std::decay<Cont>::type> make_continuation(Cont &&cont)
```

```
template<typename Cont>
    hpx::actions::continuation_impl<typename std::decay<Cont>::type> make_continuation(Cont &&f, hpx::id_type const &target)
```

```
template<typename Cont, typename F>
    std::enable_if<!std::is_same<typename std::decay<F>::type, hpx::id_type>::value, hpx::actions::continuation2_impl<typename std::decay<Cont>::type, typename std::decay<F>::type>>::type make_continuation(Cont &&cont, F &&f)
```

```
template<typename Cont, typename F>
    hpx::actions::continuation2_impl<typename std::decay<Cont>::type, typename std::decay<F>::type> make_continuation(Cont &&cont, hpx::id_type const &target, F &&f)
```

```
namespace hpx
```

```
namespace lcos
```

```
template<typename Action, typename Result>
    class packaged_action<Action, Result, false> : public hpx::lcos::promise<Result, hpx::traits::extract_action<Action>>
    Subclassed by hpx::lcos::packaged_action< Action, Result, true >
```

Public Functions

```
packaged_action()
```

```
template<typename Allocator>
    packaged_action(std::allocator_arg_t, Allocator const &alloc)
```

```
template<typename ...Ts>
    void apply(naming::id_type const &id, Ts&... vs)
```

```
template<typename ...Ts>
    void apply(naming::address &&addr, naming::id_type const &id, Ts&... vs)
```

```
template<typename Callback, typename ...Ts>
    void apply_cb(naming::id_type const &id, Callback &&cb, Ts&... vs)
```
template<typename \texttt{Callback}, typename ..\texttt{Ts}>
void \texttt{apply\_cb}(\texttt{naming::address} \&\&\texttt{addr}, \texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{Callback} \&\&\texttt{cb}, \texttt{Ts}&&... \texttt{vs})

template<typename ..\texttt{Ts}>
void \texttt{apply\_p}(\texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{threads::thread\_priority} \texttt{priority}, \texttt{Ts}&&... \texttt{vs})

template<typename ..\texttt{Ts}>
void \texttt{apply\_p}(\texttt{naming::address} \&\&\texttt{addr}, \texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{threads::thread\_priority} \texttt{priority}, \texttt{Ts}&&... \texttt{vs})

template<typename \texttt{Callback}, typename ..\texttt{Ts}>
void \texttt{apply\_p\_cb}(\texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{threads::thread\_priority} \texttt{priority}, \texttt{Callback} \&\&\texttt{cb}, \texttt{Ts}&&... \texttt{vs})

template<typename ..\texttt{Ts}>
void \texttt{apply\_deferred\_cb}(\texttt{naming::address} \&\&\texttt{addr}, \texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{Call\_back} \&\&\texttt{cb}, \texttt{Ts}&&... \texttt{vs})

Protected Types

\texttt{template<>()
using \texttt{action\_type} = \texttt{typename hpx::traits::extract\_action<}\texttt{Action}\texttt{>::type

\texttt{template<>()
using \texttt{remote\_result\_type} = \texttt{typename action\_type::remote\_result\_type

\texttt{template<>()
using \texttt{base\_type} = \texttt{promise<}\texttt{Result}, \texttt{remote\_result\_type>}

Protected Functions

\texttt{template<typename ..\texttt{Ts}>
void \texttt{do\_apply}(\texttt{naming::address} \&\&\texttt{addr}, \texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{threads::thread\_priority} \texttt{priority}, \texttt{Ts}&&... \texttt{vs})

\texttt{template<typename ..\texttt{Ts}>
void \texttt{do\_apply}(\texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{threads::thread\_priority} \texttt{priority}, \texttt{Ts}&&... \texttt{vs})

\texttt{template<typename \texttt{Callback}, typename ..\texttt{Ts}>
void \texttt{do\_apply\_cb}(\texttt{naming::address} \&\&\texttt{addr}, \texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{threads::thread\_priority} \texttt{priority}, \texttt{Callback} \&\&\texttt{cb}, \texttt{Ts}&&... \texttt{vs})

\texttt{template<typename \texttt{Callback}, typename ..\texttt{Ts}>
void \texttt{do\_apply\_cb}(\texttt{naming::id\_type} \texttt{const} \&\texttt{id}, \texttt{threads::thread\_priority} \texttt{priority}, \texttt{Callback} \&\&\texttt{cb}, \texttt{Ts}&&... \texttt{vs})

\texttt{template<typename \texttt{Action}, typename \texttt{Result}>
\texttt{class packaged\_action<}\texttt{Action, Result, true} : \texttt{public hpx::lcos::packaged\_action<}\texttt{Action, Result, false>
Public Functions

packaged_action()

Construct a (non-functional) instance of an packaged_action. To use this instance its member function apply needs to be directly called.

template<typename Allocator>
packaged_action(std::allocator_arg_t, Allocator const &alloc)

template<typename ...Ts>
void apply (naming::id_type const &id, Ts&&... vs)

template<typename ...Ts>
void apply (naming::address &&addr, naming::id_type const &id, Ts&&... vs)

template<typename Callback, typename ...Ts>
void apply_cb (naming::id_type const &id, Callback &&cb, Ts&&... vs)

template<typename Callback, typename ...Ts>
void apply_cb (naming::address &&addr, naming::id_type const &id, Callback &&cb, Ts&&... vs)

Private Types

template<>
using action_type = typename packaged_action::action_type

template<>
class promise<void, hpx::util::unused_type> : public detail::promise_base<void, hpx::util::unused_type, detail::promise_data<void>>

Public Functions

promise()

constructs a promise object and a shared state.

template<typename Allocator>
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)

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()

Abandons any shared state.

promise &operator=(promise &&other)

Abandons any shared state (30.6.4) and then as if promise(std::move(other)).swap(*this).

Return *this.
void swap (promise &other)
   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.

void set_value()
   atomically stores the value r in the shared state and makes that state ready (30.6.4).

Exceptions
   • 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

template<>
using base_type = detail::promise_base<void, hpx::util::unused_type, detail::promise_data<void>>

template<typename R, typename Allocator>
struct uses_allocator<hpx::lcos::promise<R>, Allocator> : public true_type
#include <promise.hpp> Requires: Allocator shall be an allocator (17.6.3.5)

namespace hpx

namespace lcos

Functions

template<typename Result, typename RemoteResult>
void swap (promise<Result, RemoteResult> &x, promise<Result, RemoteResult> &y)

template<>
class promise<void, hpx::util::unused_type> : public detail::promise_base<void, hpx::util::unused_type, detail::promise_data<void>>

Public Functions

promise ()
   constructs a promise object and a shared state.

template<typename Allocator>
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)
   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()
    Abandons any shared state.

promise &operator=(promise &&other)
    Abandons any shared state (30.6.4) and then as if promise(std::move(other)).swap(*this).

    **Return** *this.

    ```
    void swap(promise &other)
    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.

    ```
    void set_value()
    atomically stores the value r in the shared state and makes that state ready (30.6.4).
    ```

    **Exceptions**
    
    • 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**
    
    template<>
    using base_type = detail::promise_base<void, hpx::util::unused_type, detail::promise_data<void>>

    namespace std

    template<typename R, typename Allocator>
    struct uses_allocator<hpx::lcos::promise<R>, Allocator> : public true_type
    #include <promise.hpp> Requires: Allocator shall be an allocator (17.6.3.5)

    namespace hpx

    namespace actions

    struct set_lco_value_continuation

    **Public Functions**

    template<
    typename T>
    T operator() (naming::id_type const &lco, T &&t) const

    struct set_lco_value_unmanaged_continuation
Public Functions

```cpp
template<typename T>
T operator()(naming::id_type const &lco, T &&t) const
```

namespace hpx

Functions

```cpp
template<typename Action, typename F, typename ...Ts>
auto sync(F &&f, Ts&&... ts)
```

namespace hpx

namespace actions

Functions

```cpp
template<typename Result, typename RemoteResult, typename F, typename ...Ts>
void trigger(typed_continuation<Result, RemoteResult> &&&cont, F &&&f, Ts&&... vs)
```

```cpp
template<typename Result, typename F, typename ...Ts>
void trigger(typed_continuation<Result, util::unused_type> &&&cont, F &&&f, Ts&&... vs)
```

namespace hpx

Functions

```cpp
void trigger_lco_event(naming::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`.

```cpp
void trigger_lco_event(naming::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** (**naming::id_type** const &id, **naming::address** &&addr, **naming::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.

void **trigger_lco_event** (**naming::id_type** const &id, **naming::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.

```cpp
template<typename **Result**>
void **set_lco_value** (**naming::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.

```cpp
template<typename **Result**>
std::enable_if<!std::is_same<**typename std::decay<**Result**>::type, **naming::address**>::value>::type **set_lco_value** (**naming::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.

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.
- 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.
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`.

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(naming::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(naming::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.

void set_lco_error(naming::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.

void set_lco_error(naming::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(naming::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.
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.

```cpp
void set_lco_error(naming::id_type const &id, naming::address &&addr, std::exception_ptr &&e, naming::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.

```cpp
void set_lco_error(naming::id_type const &id, std::exception_ptr const &e, naming::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.

```cpp
void set_lco_error(naming::id_type const &id, std::exception_ptr &&e, naming::id_type const &cont, bool move_credits = true)
```

Set the error state for the LCO referenced by the given id.

namespace hpx
Functions

template<typename Action, typename ...Ts>
bool apply_p (naming::id_type const &id, threads::thread_priority priority, Ts&&... vs)

template<typename Action, typename Client, typename Stub, typename ...Ts>
bool apply_p (components::client_base<Client, Stub> const &c, threads::thread_priority priority, Ts&&... vs)

template<typename Action, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_p (DistPolicy const &policy, threads::thread_priority priority, Ts&&... vs)

template<typename Action, typename ...Ts>
bool apply (naming::id_type const &id, Ts&&... vs)

template<typename Action, typename Client, typename Stub, typename ...Ts>
bool apply (components::client_base<Client, Stub> const &c, Ts&&... vs)

template<typename Action, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply (DistPolicy const &policy, Ts&&... vs)

template<typename Action, typename Continuation, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply_p (Continuation &&c, naming::id_type const &gid, threads::thread_priority priority, Ts&&... vs)

template<typename Action, typename Continuation, typename Client, typename Stub, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply_p (Continuation &&cont, components::client_base<Client, Stub> const &c, threads::thread_priority priority, Ts&&... vs)

template<typename Action, typename Continuation, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_p(Continuation&& c, DistPolicy const& policy, threads::thread_priority priority, Ts&&... vs)

template<typename Action, typename Continuation, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply (Continuation &&&c, naming::id_type const &gid, Ts&&... vs)

template<typename Action, typename Continuation, typename Client, typename Stub, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value, bool>::type apply (Continuation &&&cont, components::client_base<Client, Stub> const &c, Ts&&... vs)

template<typename Action, typename Continuation, typename DistPolicy, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value && traits::is_continuation<Continuation>::value, bool>::type apply_p(Continuation&& c, DistPolicy const& policy, Ts&&... vs)

template<typename Action, typename ...Ts>
bool apply_c_p (naming::id_type const &contgid, naming::id_type const &gid, threads::thread_priority priority, Ts&&... vs)

template<typename Action, typename ...Ts>
bool apply_c (naming::id_type const &contgid, naming::id_type const &gid, Ts&&... vs)

template<typename Component, typename Signature, typename Derived, typename ...Ts>
bool apply_c (hpx::actions::basic_action<Component, Signature, Derived>, naming::id_type const &contgid, naming::id_type const &gid, Ts&&... vs)

namespace hpx

2.8. API reference 1357
**Functions**

template<typename \texttt{Action}, typename \texttt{Callback}, typename ...\texttt{Ts}>
bool \texttt{apply\_p\_cb} (\texttt{std::enable\_if<\texttt{traits::is\_distribution\_policy<\texttt{DistPolicy}>::value, \texttt{bool}>::type}\texttt{DistPolicy\_const\_policy, \texttt{DistPolicy\_const\_policy}}
&\texttt{policy}, \texttt{DistPolicy\_const\_policy} \texttt{cb}\texttt{, \texttt{Ts}... \texttt{vs})

template<typename \texttt{Component}, typename \texttt{Signature}, typename \texttt{Derived}, typename \texttt{Callback}, typename ...\texttt{Ts}>
bool \texttt{apply\_cb} (\texttt{std::enable\_if<\texttt{traits::is\_distribution\_policy<\texttt{DistPolicy}>::value, \texttt{bool}>::type}\texttt{DistPolicy\_const\_policy, \texttt{DistPolicy\_const\_policy}}
&\texttt{policy}, \texttt{DistPolicy\_const\_policy} \texttt{cb}\texttt{, \texttt{Ts}... \texttt{vs})

---

**Chapter 2. What’s so special about HPX?**
std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_p_cb(Continuation&& c, DistPolicy const& policy, threads::thread_priority priority, Callback&& cb, Ts&&... vs)

template<typename Action, typename Continuation, typename DistPolicy, typename Callback, typename ...Ts>
std::enable_if<traits::is_continuation<Continuation>::value && traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_cb(Continuation&& c, DistPolicy const& policy, Callback&& cb, Ts&&... vs)

template<typename Component, typename Continuation, typename Signature, typename Derived, typename DistPolicy, typename Callback, typename ...Ts>
std::enable_if<traits::is_distribution_policy<DistPolicy>::value, bool>::type apply_cb(Continuation&& c, hpx::actions::basic_action<Component, Signature, Derived> const &policy, Callback &&cb, Ts&&... vs)

template<typename Action, typename Callback, typename ...Ts>
bool apply_c_p_cb(naming::id_type const &contgid, naming::id_type const &gid, threads::thread_priority priority, Callback &&cb, Ts&&... vs)

template<typename Action, typename Callback, typename ...Ts>
bool apply_c_cb(naming::id_type const &contgid, naming::id_type const &gid, Callback &&cb, Ts&&... vs)

template<typename Action, typename Callback, typename ...Ts>
bool apply_c_p_cb(naming::id_type const &contgid, naming::address &&addr, naming::id_type const &gid, threads::thread_priority priority, Callback &&cb, Ts&&... vs)

template<typename Action, typename Callback, typename ...Ts>
bool apply_c_p_cb(naming::id_type const &contgid, naming::address &&addr, naming::id_type const &gid, threads::thread_priority priority, Callback &&cb, Ts&&... vs)
bool apply_c_cb(naming::id_type const &contgid, naming::address &addr, naming::id_type const &gid, Callback &&cb, Ts&&... vs)

namespace functional

Functions

template<typename Action, typename Callback, typename ...Ts>
apply_c_p_cb_impl<Action, typename std::decay<Callback>::type, typename std::decay<Ts>::type...> apply_c_p_cbs

template<typename Action, typename Callback, typename ...Ts>
struct apply_c_p_cb_impl

Public Types

typedef hpx::tuple<Ts...> tuple_type

Public Functions

template<typename ...Ts>
apply_c_p_cb_impl(naming::id_type const &contid, naming::address &addr, naming::id_type const &id, threads::thread_priority p, Callback &&cb, Ts&&... vs)

apply_c_p_cb_impl(apply_c_p_cb_impl &&rhs)

apply_c_p_cb_impl &operator=(apply_c_p_cb_impl &&rhs)

void operator() ()
Protected Functions

template<\texttt{std::size_t... Ts}> 
void \texttt{apply\_action} (\texttt{util::index\_pack\<Ts...\>})

Private Members

\begin{verbatim}
namespace naming
  id_type contid_
  address addr_
  id_type id_

namespace threads
  thread_priority p_

Callback cb_

tuple_type args_
\end{verbatim}

namespace \texttt{hpx}

Functions

template<\texttt{typename Action, typename Cont, \texttt{typename ...Ts}> 
bool \texttt{apply\_continue} (\texttt{Cont \&\&cont, id\_type \texttt{const \& gid, Ts\&\&... vs})

template<\texttt{typename Component, \texttt{typename Signature, \texttt{typename Derived, \texttt{typename Cont, \texttt{typename ...Ts}> 
bool \texttt{apply\_continue} (\texttt{hpx::actions::basic\_action\<Component, Signature, Derived\>, Cont \&\&cont, 
  id\_type \texttt{const \& gid, Ts\&\&... vs})

namespace \texttt{hpx}

Functions

template<\texttt{typename Action, \texttt{typename Cont, \texttt{typename Callback, \texttt{typename ...Ts}> 
bool \texttt{apply\_continue\_cb} (\texttt{Cont \&\&cont, id\_type \texttt{const \& gid, Callback \&\&cb, Ts\&\&... vs})

namespace \texttt{hpx}
Functions

template<typename Action, typename Cont, typename ...Ts>
bool apply_continue (Cont &&cont, naming::id_type const &gid, Ts&... vs)

template<typename Component, typename Signature, typename Derived, typename Cont, typename ...Ts>
bool apply_continue (hpx::actions::basic_action<Component, Signature, Derived>, Cont &&&cont, naming::id_type const &gid, Ts&... vs)

template<typename Action, typename ...Ts>
bool apply_continue (naming::id_type const &cont, naming::id_type const &gid, Ts&... vs)

template<typename Component, typename Signature, typename Derived, typename ...Ts>
bool apply_continue (hpx::actions::basic_action<Component, Signature, Derived>, naming::id_type const &cont, naming::id_type const &gid, Ts&... vs)

namespace hpx

namespace applier

Functions

template<typename Arg0>
void trigger (naming::id_type const &k, Arg0 &&arg0)

void trigger (naming::id_type const &k)

void trigger_error (naming::id_type const &k, std::exception_ptr const &e)

void trigger_error (naming::id_type const &k, std::exception_ptr &&e)

template<typename Result, typename RemoteResult>
struct action_trigger_continuation<actions::typed_continuation<Result, RemoteResult>>

Public Static Functions

template<typename F, typename ...Ts>
static void call (actions::typed_continuation<Result, RemoteResult> &&cont, F &&f, Ts&... ts)

namespace hpx

namespace traits

template<typename Result, typename RemoteResult>
struct action_trigger_continuation<actions::typed_continuation<Result, RemoteResult>>
Public Static Functions

template<typename F, typename ... Ts>
static void call (actions::typed_continuation<Result, RemoteResult> &cont, F &&f, 
Ts&&... ts)

checkpoint

The contents of this module can be included with the header hpx/modules/checkpoint.hpp. These headers 
may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these 
at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the 
module header hpx/modules/checkpoint.hpp, not the particular header in which the functionality you would 
like to use is defined. See Public API for a list of names 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

\texttt{std::ostream \& operator\ll< (std::ostream \& ost, checkpoint const \& ckp) \}
\texttt{Operator\ll 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\ll 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!

\textbf{Parameters}
- \texttt{ost}: Output stream to write to.
- \texttt{ckp}: Checkpoint to copy from.

\textbf{Return} Operator\ll returns the ostream object.

\texttt{std::istream \& operator\gg< (std::istream \& ist, checkpoint \& ckp) \}
\texttt{Operator\gg 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\ll overload. Be 
mindful of this additional read and write when you use different facilities to read in or write out data 
from a checkpoint!

\textbf{Parameters}
- \texttt{ist}: Input stream to write from.
- \texttt{ckp}: Checkpoint to write to.

\textbf{Return} Operator\gg 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 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.

**Return** 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.

**Return** 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> 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.

**Return** 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.

```cpp
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.

**Return** 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.

```cpp
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.

**Return** `Save_checkpoint` which is passed `hpx::launch::sync_policy` will return a checkpoint which contains the serialized values checkpoint.

```cpp
template<typename T, typename ...Ts>
checkpoint save_checkpoint (hpx::launch::sync_policy sync_p, checkpoint &&c, T &&t, Ts&&... ts)
```

`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.

**Return** `Save_checkpoint` which is passed `hpx::launch::sync_policy` will return a checkpoint which contains the serialized values checkpoint.

```cpp
template<
typename T, typename ...Ts>
typename U = typename std::enable_if<!hpx::traits::is_launch_policy<T>::value &
  hpx::future<checkpoint> prepare_checkpoint (T const &t, Ts const&... ts)
```

`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.

**Return** `prepare_checkpoint` returns a properly resized checkpoint object that can be used for a subsequent `restore_checkpoint` operation.

**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.

```cpp
template<typename T, typename ...Ts>
hpx::future<checkpoint> prepare_checkpoint (checkpoint &c, T const &t, Ts const&... ts)
```

**Return**

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.

```cpp
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)
```

**Return**

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.

```cpp
template<typename T, typename ...Ts>
hpx::future<checkpoint> prepare_checkpoint (hpx::launch p, checkpoint &&c, T const &t, Ts const&... ts)
```

**Return**

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.

**Return** prepare_checkpoint returns a properly resized checkpoint object that can be used for a subsequent restore_checkpoint operation.

**Template Parameters**
- \( T \): A container to restore.
- \( T_s \): 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.
- \( t_s \): Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

```cpp
template<typename \( T \), typename ...Ts>
void restore_checkpoint (checkpoint const &c, T &t, Ts&... ts)
```

**class checkpoint**

includes `<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

```cpp
using const_iterator = std::vector::const_iterator
```

Public Functions

```cpp
checkpoint ()
~checkpoint ()
checkpoint (checkpoint const &c)
checkpoint (checkpoint &&c)
checkpoint (std::vector<char> const &vec)
checkpoint (std::vector<char> &&vec)
checkpoint &operator= (checkpoint const &c)
checkpoint &operator= (checkpoint &&c)
const_iterator begin () const
const_iterator end () const
std::size_t size () const
char *data ()
char const *data () const
```

Private Functions

```cpp
template<typename Archive>
void serialize (Archive &arch, const unsigned int)
```

Private Members

```cpp
std::vector<char> data_
```

Friends

```cpp
friend hpx::util::hpx::serialization::access
```

```cpp
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.

**Return**  
Operator<< returns the ostream object.

```cpp
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.

**Return**  
Operator>> returns the ostream object.

```cpp
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.

**Return**  
Restore_checkpoint returns void.

**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.

```cpp
bool operator==(checkpoint const &lhs, checkpoint const &rhs)

bool operator!=(checkpoint const &lhs, checkpoint const &rhs)
```

**checkpoint_base**

The contents of this module can be included with the header `hpx/modules/checkpoint_base.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/checkpoint_base.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

**namespace hpx**
namespace util

## Functions

```cpp
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**
- `cont`: Container instance used to store the checkpoint data
- `ts`: Variable instances to be inserted into the checkpoint.

```cpp
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.

```cpp
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
The contents of this module can be included with the header `hpx/modules/collectives.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/collectives.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace collectives

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.

Return 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.

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).
- 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.
- 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_gather support object. This value is optional and defaults to ‘0’ (zero).

hpx::future<std::vector<std::decay_t<T>>> all_gather(communicator comm, T &&result, this_site_arg this_site = this_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.

Return 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.

Parameters

- comm: A communicator object returned from create_reducer
- 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.

namespace hpx

namespace collectives

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.

Return 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.

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.
  • 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.

namespace hpx

namespace collectives

2.8. API reference 1373
Functions

```cpp
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.

**Return** 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.

**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.
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to ‘0’ (zero).

```cpp
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())
```

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.

**Return** 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.

**Parameters**
- `comm`: A communicator object returned from `create_reducer`
- `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.

```
namespace hpx

namespace collectives

struct generation_arg
```
Public Functions

```cpp
cconstexpr generation_arg (std::size_t generation = std::size_t(-1))

cconstexpr generation_arg &operator= (std::size_t generation)

cconstexpr operator std::size_t() const
```

Public Members

```cpp
std::size_t generation_
```

struct num_sites_arg

Public Functions

```cpp
cconstexpr num_sites_arg (std::size_t num_sites = std::size_t(-1))

cconstexpr num_sites_arg &operator= (std::size_t num_sites)

cconstexpr operator std::size_t() const
```

Public Members

```cpp
std::size_t num_sites_
```

struct root_site_arg

Public Functions

```cpp
cconstexpr root_site_arg (std::size_t root_site = std::size_t(0))

cconstexpr root_site_arg &operator= (std::size_t root_site)

cconstexpr operator std::size_t() const
```

Public Members

```cpp
std::size_t root_site_
```

struct tag_arg

Public Functions

```cpp
cconstexpr tag_arg (std::size_t tag = std::size_t(0))

cconstexpr tag_arg &operator= (std::size_t tag)

cconstexpr operator std::size_t() const
```
namespace hpx

namespace lcos

class barrier

#include <barrier.hpp> The barrier is an implementation performing a barrier over a number of participating threads. The different threads don’t have to be on the same locality. This barrier can be invoked in a distributed application.

For a local only barrier
See hpx::lcos::local::barrier.
Public Functions

**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
- rank: The rank of the calling site for this invocation

**void** **wait**( )

Wait until each participant entered the barrier. Must be called by all participants

**Return** This function returns once all participants have entered the barrier (have called wait).

**hpx::future<void>** **wait**(hpx::launch::async_policy)

Wait until each participant entered the barrier. Must be called by all participants

**Return** a future that becomes ready once all participants have entered the barrier (have called wait).
Public Static Functions

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 locali-
ties.

namespace hpx

namespace collectives

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 = genera-
tion_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.

Return: This function returns a future that will become ready once the broadcast operation has been
completed.

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.
• 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.

template<typename T>

hpx::future<void> broadcast_to (communicator comm, 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.

Return: This function returns a future that will become ready once the broadcast operation has been
completed.

Parameters
• comm: A communicator object returned from create_reduser
• 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.
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.

**Return** 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.

**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.
- **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.

template<typename T>

hpx::future<T> broadcast_from(communicator comm, this_site_arg this_site = this_site_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.

**Return** 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.

**Parameters**
- **comm**: A communicator object returned from create_reducer
- **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.

namespace hpx

namespace lcos

**Functions**

template<typename Action, typename ArgN, ...>hpx::future<std::vector<decltype(Action(hpx::id_type, ArgN, ...))> > hpx::lcos::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.

**Return** This function returns a future representing the result of the overall reduction operation.

**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.
cpp
template<typename Action, typename ArgN, ...> void hpx::lcos::broadcast_apply(std::vector<hpx::id_type> const & ids, ArgN argN, ...)

Perform an asynchronous (fire&forget) distributed broadcast operation.

The function hpx::lcos::broadcast_apply 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.

cpp
template<typename Action, typename ArgN, ...> hpx::future<std::vector<decltype(Action(hpx::id_type, ArgN, ..., std::size_t))>> hpx::lcos::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.

**Return**
This function returns a future representing the result of the overall reduction operation.

**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.

cpp
template<typename Action, typename ArgN, ...> void hpx::lcos::broadcast_apply_with_index(std::vector<hpx::id_type> const & ids, ArgN argN, ...)

Perform an asynchronous (fire&forget) distributed broadcast operation.

The function hpx::lcos::broadcast_apply_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.
• argvN: [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

namespace hpx

namespace collectives

Functions

hpx::future<channel_communicator> create_channel_communicator(const char* 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 function 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.

Return This function returns a future to a new communicator object usable with the collective operation.

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.

channel_communicator create_channel_communicator (hpx::launch::sync_policy, char* 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 function 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.

Return This function returns a new communicator object usable with the collective operation.

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.

template<typename T>
hpx::future<void> set(channel_communicator comm, that_site_arg site, T &&value, tag_arg tag = 0)

Send a value to the given site

This function sends a value to the given site based on the given communicator.
This function returns a future<void> that becomes ready once the data transfer operation has finished.

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

```cpp
template<typename T>
hpx::future<T> get (channel_communicator comm, that_site_arg site, tag_arg tag = 0)
```

This function receives a value from the given site based on the given communicator.

**Return** This function returns a future<T> that becomes ready once the data transfer operation has finished. The future will hold the received value.

**Parameters**
- **comm**: The channel communicator object to use for the data transfer
- **site**: The source site

```cpp
namespace hpx

namespace lcos

Functions

hpx::future<hpx::id_type> create_communication_set (char const *basename, std::size_t num_sites = std::size_t(-1),
std::size_t this_site = std::size_t(-1),
std::size_t arity = std::size_t(-1))
```

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).

**Return** This function returns a future holding an id_type of the communicator object to be used on the current locality.

**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.
- **arity**: The number of children each of the communication nodes is connected to (default: picked based on `num_sites`)

```cpp
namespace hpx

namespace collectives
```
Functions

communicator **create_communicator**(char *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 function 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.).

**Return**
This function returns a new communicator object usable with the collective operation.

**Parameters**
- basename: The base name identifying the collective operation.
- num_sites: The number of participating sites (default: all localities).
- 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.
- 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.
- root_site: The site that is responsible for creating the collective support object. This value is optional and defaults to ‘0’ (zero).

namespace hpx

namespace collectives

Functions

```
template<
    typename T, typename F>

hpx::future<std::decay_t<T>> **exclusive_scan**(char const *basename, T &local_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 thje root_site.

**Return**
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.

**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).
- 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 sup-
plied only if the exclusive_scan operation on the given base name has to be performed more than once.

- **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.

  - **root_site**: The site that is responsible for creating the exclusive_scan support object. This value is optional and defaults to ‘0’ (zero).

  

```cpp
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())
```

Ex 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 thje root_site.

**Return** 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.

**Parameters**

- **comm**: A communicator object returned from **create_reducer**
- **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.

namespace hpx

namespace lcos

**Functions**

```cpp
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...>hpx::future<decltype(Action(hpx::id_type, ArgN, ...))> hpx::lcos::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.

**Return** This function returns a future representing the result of the overall folding operation.

**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.
template<typename Action, typename FoldOp, typename Init, typename ArgN, ...> hpx::future<decltype(Action(hpx::id_type, ...))> hpx::lcos::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.

**Return** This function returns a future representing the result of the overall folding operation.

**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.

---

template<typename Action, typename FoldOp, typename Init, typename ArgN, ...> hpx::future<decltype(Action(hpx::id_type, ArgN, ...))> hpx::lcos::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.

**Return** This function returns a future representing the result of the overall folding operation.

**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.

---

template<typename Action, typename FoldOp, typename Init, typename ArgN, ...> hpx::future<decltype(Action(hpx::id_type, ... 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.
The type of the initial value must be convertible to the result type returned from the invoked action.

This function returns a future representing the result of the overall folding operation.

**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.

```cpp
namespace hpx

namespace collectives

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.

**Return** This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

**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).
- **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.
- **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.

```
• 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.

```cpp
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)

**Return** This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

**Parameters**
- basename: The base name identifying the gather operation
- 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.
- 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.
- root_site: The sequence number of the central gather point (usually the locality id). This value is optional and defaults to 0.

```cpp
template<typename T>
hpx::future<
    std::vector<
decay_t<T>>>
gather_there(
    communicator comm,
    T &&result,
    this_site_arg this_site = this_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)

**Return** This function returns a future holding a vector with all gathered values. It will become ready once the gather operation has been completed.

**Parameters**
- comm: A communicator object returned from `create_reducer`
- 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.

```cpp
namespace hpx
```

```cpp
namespace collectives
```
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.

**Return** 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.

**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).
- `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.
- `this_site`: The sequence number of this invocation (usually the locality id). This value is optional and defaults to '0' (zero).

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.

**Return** 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.

**Parameters**
- `comm`: A communicator object returned from `create_reducer`
- `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.

namespace hpx

namespace lcos

class latch : public components::client_base<latch, lcos::server::latch>
Public Functions

latch()

latch (std::ptrdiff_t count)
  Initialize the latch
  Requires: count >= 0. Synchronization: None Postconditions: counter_ == count.

latch (naming::id_type const &id)
  Extension: Create a client side representation for the existing server::latch instance with the given
global id id.

latch (hpx::future<naming::id_type> &&f)
  Extension: Create a client side representation for the existing server::latch instance with the given
global id id.

latch (hpx::shared_future<naming::id_type> const &id)
  Extension: Create a client side representation for the existing server::latch instance with the given
global id id.

latch (hpx::shared_future<naming::id_type> &&id)

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.

Exceptions
  • Nothing.

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.

Exceptions
  • Nothing.

bool is_ready () const
  Returns: counter_ == 0. Does not block.

Exceptions
  • Nothing.

void wait () const
  If counter_ is 0, returns immediately. Otherwise, blocks the calling thread at the synchronization
  point until counter_ reaches 0.

Exceptions
  • Nothing.
Private Types

typedef components::client_base<latch, lcos::server::latch> base_type

namespace hpx

namespace collectives

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.

Return 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.

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).
- 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.
- 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.

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())

Reduce a given value at the given call site

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.

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.
This function transmits the value given by `result` to a central reduce site (where the corresponding `reduce_here` is executed)

**Return**  This function returns a future<void>. It will become ready once the reduction operation has been completed.

**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.
- `generation`: The generational counter identifying the sequence number of the reduction operation performed on the given base name. This is optional and needs to be supplied only if the reduction operation on the given base name has to be performed more than once.
- `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.
- `root_site`: The sequence number of the central reduction point (usually the locality id). This value is optional and defaults to 0.

```cpp
template<typename T>
hpx::future<void> reduce_there(communicator comm, T &&local_result, this_site_arg this_site = this_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)

**Return**  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.

**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.

```cpp
namespace hpx

namespace lcos

Functions

```cpp
template<typename Action, typename ReduceOp, typename ArgN, ...>hpx::future<decltype(Action(hpx::id_type, ArgN, ...))> hpx::lcos::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).

**Return**  This function returns a future representing the result of the overall reduction operation.

**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.

```cpp
template<typename Action, typename ReduceOp, typename ArgN, ...> hpx::future<decltype(Action(hpx::id_type, ArgN, ..., std::size_t))> hpx::lcos::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.

**Return** This function returns a future representing the result of the overall reduction operation.

**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.

```cpp
namespace hpx

namespace collectives

Functions

```cpp
template<typename T>
```

```cpp
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.

**Return** This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

**Parameters**

- `basename`: The base name identifying the scatter operation
- `generation`: The generational counter identifying the sequence number of the scatter operation performed on the given base name. This is optional and needs to be supplied only if the scatter operation on the given base name has to be performed more than once.
- `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.
- `root_site`: The sequence number of the central scatter point (usually the locality id). This value is optional and defaults to 0.
hpx::future<T> scatter_from (communicator comm, this_site_arg this_site = this_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.

**Return**  This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

**Parameters**
- **comm**: A communicator object returned from create_reducer
- **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.

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)

**Return**  This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

**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 scatter operation performed on the given base name. This is optional and needs to be supplied only if the scatter operation on the given base name has to be performed more than once.
- **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.

template<typename T>
hpx::future<T> scatter_to (communicator comm, std::vector<T> &&result, this_site_arg this_site = this_site_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)

**Return**  This function returns a future holding a the scattered value. It will become ready once the scatter operation has been completed.

**Parameters**
- **comm**: A communicator object returned from create_reducer
- **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.

namespace hpx

namespace lcos

2.8. API reference 1393
Functions

template<typename F, typename ...Args>
hpx::future<void> define_spmd_block (std::string &name, std::size_t images_per_locality, F&, Args&... args)

struct spmd_block
  #include <spmd_block.hpp> The class spmd_block defines an interface for launching multiple images while giving handles to each image to interact with the remaining images. The define_spmd_block function templates create multiple images of a user-defined action and launches them in a possibly separate thread. A temporary spmd block object is created and diffused to each image. The constraint for the action given to the define_spmd_block function is to accept a spmd_block as first parameter.

Public Functions

spmd_block ()
spmd_block (std::string const &name, std::size_t images_per_locality, std::size_t num_images, std::size_t image_id)

std::size_t get_images_per_locality () const
std::size_t get_num_images () const
std::size_t this_image () const

void sync_all () const
hpx::future<void> sync_all (hpx::launch::async_policy const&) const

void sync_images (std::set<std::size_t> const &images) const
void sync_images (std::vector<std::size_t> const &input_images) const

template<typename Iterator>
std::enable_if<traits::is_input_iterator<Iterator>::value>::type sync_images (Iterator begin, Iterator end) const

template<typename I>
std::enable_if<std::all_of<typename std::tuple<std::size_t...>::value>::type sync_images (I...)

hpx::future<void> sync_images (hpx::launch::async_policy const&, std::set<std::size_t> const &images) const
hpx::future<void> sync_images (hpx::launch::async_policy const &policy, std::vector<std::size_t> const &input_images) const

template<typename Iterator>

std::enable_if<std::is_input_iterator<Iterator>::value, hpx::future<void>>::type sync_images (hpx::launch::async_policy const &policy, Iterator begin, Iterator end)

```
const
```

```cpp
template<typename ...I>
std::enable_if<std::util::all_of<std::is_integral<I>::value, hpx::future<void>>::type sync_images (I ...i)
```

**Private Types**

**using barrier_type = hpx::lcos::barrier**

**using table_type = std::map<std::set<std::size_t>, std::shared_ptr<barrier_type>>**

**Private Functions**

```
template<typename Archive>
void serialize (Archive&, unsigned)
```

**Private Members**

```
std::string name_
std::size_t images_per_locality_
std::size_t num_images_
std::size_t image_id_
hpx::util::jenkins_hash hash_
std::shared_ptr<hpx::lcos::barrier> barrier_
table_type barriers_
```
command_line_handling

The contents of this module can be included with the header hpx/modules/command_line_handling.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/command_line Handling.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace util

struct command_line_handling

Public Functions

command_line_handling (runtime_configuration rtcfg, std::vector<std::string> ini_config, function_nonser<int> hpx::program_options::variables_map &vm
> hpx_main_f

int call (hpx::program_options::options_description const &desc_cmdline, int argc, char **argv, std::vector<std::shared_ptr<components::component_registry_base>> &component_registries)

Public Members

hpx::program_options::variables_map vm_
util::runtime_configuration rtcfg_
std::vector<std::string> ini_config_
util::function_nonser<int> (hpx::program_options::variables_map &vm) > hpx_main_f_
std::size_t node_
std::size_t num_threads_
std::size_t num_cores_
std::size_t num_localities_
std::size_t pu_step_
std::size_t pu_offset_
std::string queuing_
std::string affinity_domain_
std::string affinity_bind_
```
std::size_t numa_sensitive_
bool use_process_mask_
bool cmd_line_parsed_
bool info_printed_
bool version_printed_

Protected Functions

void check_affinity_domain() const
void check_affinity_description() const
void check_pu_offset() const
void check_pu_step() const

bool handle_arguments (util::manage_config&cfmap, hpx::program_options::variables_map &vm,
                      std::vector<std::string> &ini_config, std::size_t &node,
                      bool initial = false)

void enable_logging_settings (hpx::program_options::variables_map &vm,
                              std::vector<std::string> &ini_config)

void store_command_line (int argc, char **argv)

void store_unregistered_options (std::string const &cmd_name,
                                 std::vector<std::string> const &unregistered_options)

bool handle_help_options (hpx::program_options::options_description const &help)

void handle_attach_debugger ()

std::vector<std::string> preprocess_config_settings (int argc, char **argv)

namespace hpx

namespace util

Functions

int handle_late_commandline_options (util::runtime_configuration &ini,
                                     hpx::program_options::options_description const &options,
                                     void (*handle_print_bind)) std::size_t

namespace hpx

namespace util
```
Functions

bool parse_commandline (hpx::util::section const &rtcfg, hpx::program_options::options_description const &app_options, std::string const &cmdline, hpx::program_options::variables_map &vm, std::size_t node, int error_mode = return_on_error, hpx::runtime_mode mode = runtime_mode::default_, hpx::program_options::options_description *visible = nullptr, std::vector<std::string> *unregistered_options = nullptr)

bool parse_commandline (hpx::util::section const &rtcfg, hpx::program_options::options_description const &app_options, const std::string &arg0, std::vector<std::string> const &args, hpx::program_options::variables_map &vm, std::size_t node, int error_mode = return_on_error, hpx::runtime_mode mode = runtime_mode::default_, hpx::program_options::options_description *visible = nullptr, std::vector<std::string> *unregistered_options = nullptr)

components

The contents of this module can be included with the header hpx/modules/components.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/components.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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.

Return A list of futures representing the ids which were registered using the given base name.

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.

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.
This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

**Return** A list of futures representing the ids which were registered using the given base name.

**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.

**Parameters**
- base_name: [in] The base name for which to retrieve the registered ids.
- num_ids: [in] The number of registered ids to expect.

```
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.

**Return** A list of futures representing the ids which were registered using the given base name and sequence numbers.

**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.

**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.

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** A list of futures representing the ids which were registered using the given base name and sequence numbers.

**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.

**Parameters**
- base_name: [in] The base name for which to retrieve the registered ids.
- ids: [in] The sequence numbers of the registered ids.

```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.

**Return** A representing the id which was registered using the given base name and sequence numbers.

**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.

**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.

**Return** A representing the id which was registered using the given base name and sequence numbers.

**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.

**Parameters**

- base_name: [in] The base name for which to retrieve the registered ids.
- sequence_nr: [in] The sequence number of the registered id.

**Return** A future representing the result of the registration operation itself.

**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.

```cpp
template<typename Client>
Client unregister_with_basename (std::string base_name, std::size_t sequence_nr)

Unregister the given id using 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.

Return A future representing the result of the un-registration operation itself.

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.

Return A future representing the result of the un-registration operation itself.

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.

namespace hpx

Functions

`hpx::future<bool> register_with_basename (std::string base_name, hpx::id_type 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.

Return A future representing the result of the registration operation itself.

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.

```cpp
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.

**Return** A future representing the result of the registration operation itself.

**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.

```cpp
namespace hpx

namespace components

template<typename Component>
class client : public hpx::components::client_base<client<Component>, Component>

Public Functions

client ()
client (naming::id_type const &id)
client (naming::id_type &id)
client (future_type const &f)
client (future_type &f)
client (future<naming::id_type> &f)
client (future<client> &c)
client (client const &rhs)
client (client &rhs)
client &operator= (naming::id_type const &id)
client &operator= (naming::id_type &id)
```
client &operator= (future_type const &f)
client &operator= (future_type &&f)
client &operator= (future<naming::id_type> &&f)
client &operator= (client const &rhs)
client &operator= (client &&rhs)

Private Types

template<>
using base_type = client_base<client<Component>, Component>

template<>
using future_type = typename base_type::future_type

namespace hpx

namespace components

Functions

template<typename Derived, typename Stub>
bool operator==(client_base<Derived, Stub> const &lhs, client_base<Derived, Stub> const &rhs)

template<typename Derived, typename Stub>
bool operator<(client_base<Derived, Stub> const &lhs, client_base<Derived, Stub> const &rhs)

class client_base : public detail::make_stub::type<Stub>

Public Types

template<>
using stub_argument_type = Stub

template<>
using server_component_type = typename detail::make_stub::server_component_type

template<>
using is_client_tag = void
Public Functions

client_base()
client_base(id_type const &id)
client_base(id_type &id)
client_base(shared_future<id_type> const &f)
client_base(shared_future<id_type> &&f)
client_base(client_base const &rhs)
client_base(client_base &&rhs)
client_base(future<Derived> &&d)
~client_base()
client_base &operator=(id_type const &id)
client_base &operator=(id_type &id)
client_base &operator=(shared_future<id_type> const &f)
client_base &operator=(shared_future<id_type> &&f)
client_base &operator=(future<id_type> &&f)
client_base &operator=(client_base const &rhs)
client_base &operator=(client_base &&rhs)
bool valid() const
operator bool() const
void free()
id_type const &get_id() const
naming::gid_type const &get_raw_gid() const
shared_future<id_type> detach()
shared_future<id_type> share() const
void reset(id_type const &id)
void reset(id_type &id)
void reset(shared_future<id_type> &&&rhs)
id_type const &get() const
bool is_ready() const
bool has_value() const
bool has_exception() const

void wait() const

std::exception_ptr get_exception_ptr() const

template<typename F>
    hpx::traits::future_then_result_t<Derived, F> then(launch l, F &&f)

template<typename F>
    hpx::traits::future_then_result_t<Derived, F> then(F &&f)

future<bool> register_as(std::string symbolic_name, bool manage_lifetime = true)

void connect_to(std::string const &symbolic_name)

std::string const &registered_name() const

Protected Types

template<>
    using stub_type = typename detail::make_stub<Stub>::type

template<>
    using shared_state_type = lcos::detail::future_data_base<id_type>

template<>
    using future_type = shared_future<id_type>

Protected Functions

client_base (hpx::intrusive_ptr<shared_state_type> const &state)

client_base (hpx::intrusive_ptr<shared_state_type> &&state)

Protected Attributes

hpx::intrusive_ptr<shared_state_type> shared_state_

Private Static Functions

template<typename F>
    static hpx::traits::future_then_result<Derived, F>::cont_result on_ready (shared_future<id_type> &&fut, F f)

    static bool register_as_helper (client_base const &f, std::string symbolic_name, bool manage_lifetime)
Functions

template<typename Archive, typename Derived, typename Stub>
void serialize(Archive & ar, ::hpx::components::client_base<Derived, Stub> &f, unsigned version)

namespace hpx

namespace components

template<typename Executor, typename BaseComponent>
struct executor_component : public BaseComponent

Public Functions

template<typename ...Arg>
exector_component (executor_type const & exec, Arg&&... arg)

Public Static Functions

static void execute (hpx::threads::thread_function_type const & f)

template<typename Executor_ = Executor>
static void schedule_thread (hpx::naming::address::address_type lva,
                              hpx::naming::address::component_type,
                              hpx::threads::thread_init_data &data,
                              hpx::threads::thread_schedule_state)

This is the default hook implementation for schedule_thread which forwards to the executor instance associated with this component.

Protected Attributes

exector_type exec_

Private Types

typedef BaseComponent base_type

typedef Executor executor_type

typedef base_type::this_component_type this_component_type

namespace hpx
Functions

```cpp
template<typename Component>
hpx::future<std::shared_ptr<Component>> get_ptr (naming::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.
```

**Return** This function returns a future representing the pointer to the underlying memory for the component instance with the given `id`.

**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`: only template parameter has to be the type of the server side component.

```cpp
template<typename Derived, typename Stub>
hpx::future<std::shared_ptr<typename components::client_base<Derived, Stub>::server_component_type>> get_ptr (components::client_base<Derived, Stub> const &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.
```

**Return** This function returns a future representing the pointer to the underlying memory for the component instance with the given `id`.

**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.

```cpp
template<typename Component>
std::shared_ptr<Component> get_ptr (launch::sync_policy p, naming::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.
```

2.8. API reference
Return This function returns the pointer to the underlying memory for the component instance with the given id.

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 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.

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

- The: only template parameter has to be the type of the server side component.

```cpp
template<typename Derived, typename Stub>
std::shared_ptr<typename components::client_base<Derived, Stub>::server_component_type> get_ptr(launch::sync_policy p, components::client_base<Derived, Stub> 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.

Return This function returns the pointer to the underlying memory for the component instance with the given id.

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.

```cpp
namespace hpx

namespace components

Functions

template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::id_type const &id)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::id_type &&id)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::future<hpx::id_type> const &id)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::future<hpx::id_type> &&id)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::shared_future<hpx::id_type> const &id)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, Client>::type make_client (hpx::shared_future<hpx::id_type> &&id)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::id_type> const &ids)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::id_type> &&ids)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::future<hpx::id_type>> const &ids)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::future<hpx::id_type>> &&ids)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::shared_future<hpx::id_type>> const &ids)

template<typename Client>
std::enable_if<traits::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::shared_future<hpx::id_type>> &&ids)
```

2.8. API reference
std::enable_if<std::is_client<Client>::value, std::vector<Client>>::type make_clients (std::vector<hpx::shared_future<hpx::id_type>>& ids)

components_base

The contents of this module can be included with the header hpx/modules/components_base.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/components_base.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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, naming::id_type const &id, error_code &ec = throws)

hpx::future<bool> register_name (std::string const &name, naming::id_type const &id)

naming::id_type unregister_name (launch::sync_policy, std::string const &name, error_code &ec = throws)

hpx::future<naming::id_type> unregister_name (std::string const &name)

naming::id_type resolve_name (launch::sync_policy, std::string const &name, error_code &ec = throws)

hpx::future<naming::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)

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)
HPX Documentation, master

```cpp
hpx::naming::gid_type get_locality()

std::vector<std::uint32_t> get_all_locality_ids(naming::component_type type, error_code &ec = throws)

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::id_type const &id, error_code &ec = throws)

bool is_local_address_cached(naming::id_type const &id, naming::address &addr, 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)

bool is_local_lva_encoded_address(naming::id_type const &id)

hpx::future<naming::address> resolve(naming::id_type const &id)

naming::address resolve(launch::sync_policy, naming::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 (naming::id_type const &id, error_code &ec = throws)
    Invoke an asynchronous garbage collection step on the given target locality.

void garbage_collect (naming::id_type const &id, error_code &ec = throws)
    Invoke a synchronous garbage collection step on the given target locality.

naming::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<std::int64_t> incref (naming::gid_type const &gid, std::int64_t credits, naming::id_type const &keep_alive = naming::invalid_id)

std::int64_t incref (launch::sync_policy, naming::gid_type const &gid, std::int64_t credits = 1, naming::id_type const &keep_alive = naming::invalid_id, error_code &ec = throws)

std::int64_t replenish_credits (naming::gid_type &gid)

hpx::future<naming::id_type> get_colocation_id (naming::id_type const &id)

naming::id_type get_colocation_id (launch::sync_policy, naming::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<naming::id_type, naming::address>> begin_migration (naming::id_type const &id)

bool end_migration (naming::id_type const &id)

hpx::future<void> mark_as_migrated (naming::gid_type const &gid, util::unique_function_nonser<std::pair<bool, hpx::future<void>>>)

std::pair<bool, components::pinned_ptr> was_object_migrated (naming::gid_type const &gid, util::unique_function_nonser<components::pinned_ptr>)

void unmark_as_migrated (naming::gid_type const &gid)

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)

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
#else include <component_commandline.hpp> The component_startup_shutdown provides a minimal implementation of a component’s startup/shutdown function provider.

Public Functions

hp::program_options::options_description add_commandline_options()  
Return any additional command line options valid for this component.

Return The module is expected to fill a options_description object with any additional command line options this component will handle.
Note This function will be executed by the runtime system during system startup.

namespace commandline_options_provider

Functions

hp::program_options::options_description add_commandline_options()

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_function_type &, bool &)> Startup, bool(*)(shutdown_function_type &, bool &)> Shutdown> hpx::components::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

bool get_startup_function(startup_function_type &startup, bool &pre_startup)
Return any startup function for this component.

Return Returns true if the parameter startup has been successfully initialized with the startup function.
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.

bool get_shutdown_function(shutdown_function_type &shutdown, bool &pre_shutdown)
Return any startup function for this component.

Return Returns true if the parameter shutdown has been successfully initialized with the shutdown function.
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.

Defines

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

```cpp
using component_deleter_type = void (*)(hpx::naming::gid_type const &, hpx::naming::address const &);
```

Enums

```cpp
enum component_enum_type
{
  component_invalid = naming::address::component_invalid,
  component_runtime_support = 0,
  component_plain_function = 1,
  component_base_lco = 2,
  component_base_lco_with_value_unmanaged = 3,
  component_base_lco_with_value = 4,
  component_latch = ((5 << 10) | component_base_lco_with_value),
  component_barrier = ((6 << 10) | component_base_lco),
  component_promise = ((7 << 10) | component_base_lco_with_value),
  component_agas_locality_namespace = 8,
  component_agas_primary_namespace = 9,
  component_agas_component_namespace = 10,
  component_agas_symbol_namespace = 11,
  component_last,
  component_first_dynamic = component_last,
  component_upper_bound = 0xffffL;
}
``` 

```cpp
enum factory_state_enum
{
  factory_enabled = 0,
  factory_disabled = 1,
  factory_check = 2,
}
``` 

Functions

```cpp
bool &enabled(component_type type);
util::atomic_count &instance_count(component_type type);
component_deleter_type &deleter(component_type type);
bool enumerate_instance_counts(std::unique_function_nonser<bool> const &f, component_type &);
```
const 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)  
    The lower short word of the component type is the type of the component exposing the actions.

constexpr component_type getDerived_type (component_type t)  
    The upper short word of the component is the actual component type.

constexpr component_type derived_component_type (component_type derived, component_type base)  
    A component derived from a base component exposing the actions needs to have a specially formatted component type.

cconstexpr bool types_are_compatible (component_type lhs, component_type rhs)  
    Verify the two given component types are matching (compatible)

template<typename Component, typename Enable = void>  
    constexpr char const *get_component_name ()

template<typename Component, typename Enable = void>  
    constexpr const char *get_component_base_name ()

template<typename Component>  
    component_type get_component_type ()

template<typename Component>  
    void set_component_type (component_type type)

namespace naming

Functions naming

std::ostream &operator<< (std::ostream&, address const&)

namespace hpx

namespace components

Typedefs
typedef component_base<Component> instead
template<typename Component>  
using abstract_simple_component_base = abstract_component_base<Component>

template<typename Component, typename Derived>  
class managed_component
    #include "managed_component_base.hpp" The managed_component template is used as a 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.
namespace hpx

namespace util

class unique_id_ranges
#include <generate_unique_ids.hpp> The unique_id_ranges class is a type responsible for generating unique ids for components, parcels, threads etc.

Public Functions

unique_id_ranges ()

naming::gid_type get_id (std::size_t count = 1)  
Generate next unique component id.

void set_range (naming::gid_type const &lower, naming::gid_type const &upper)

Private Types

enum [anonymous]
size of the id range returned by command_getidrange FIXME: is this a policy?

Values:

range_delta = 0x100000

typedef hpx::util::spinlock mutex_type

Private Members

mutex_type mtx_

naming::gid_type lower_
  The range of available ids for components.

naming::gid_type upper_

template< typename Component>
struct get_lva<Component, typename std::enable_if<!traits::is_managed_component<Component>::value>::type>
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 Component *call (naming::address_type lva)


Public Static Functions

\texttt{static Component \*call(naming:address\_type lva)}

\texttt{template<typename Component> struct get\_lva<Component, typename std::enable_if<traits::is\_managed\_component<Component>:\:value && std::is\_const<Component>:\:value>::type>}

Public Static Functions

\texttt{static Component \*call(naming:address\_type lva)}

\texttt{template<typename Component> struct create\_helper<Component, typename std::enable_if<traits::component\_decorates\_action<Component>:\:value>::type>}

Public Static Functions

\texttt{static pinned\_ptr call(naming:address\_type lva)}

namespace hpx

namespace components

class pinned\_ptr

Public Functions

\texttt{pinned\_ptr()}\n\texttt{pinned\_ptr(pinned\_ptr const \&rhs)}\n\texttt{pinned\_ptr(pinned\_ptr \&\&rhs)}\n\texttt{pinned\_ptr \&operator=(pinned\_ptr const \&rhs)}\n\texttt{pinned\_ptr \&operator=(pinned\_ptr \&\&rhs)}

Public Static Functions

\texttt{template<typename Component> static pinned\_ptr create(naming:address\_type lva)}

Private Functions

\texttt{template<typename Component> pinned\_ptr(naming:address\_type lva, id<Component>)}
### Private Members

```cpp
std::unique_ptr<detail::pinned_ptr_base> data_
```

```cpp
template<typename Component, typename Enable = void>
struct create_helper
```

### Public Static Functions

```cpp
static pinned_ptr call (naming::address_type)
```

```cpp
template<typename Component>
struct create_helper<Component, typename std::enable_if<traits::component_decorates_action<Component>>::value>::type
```

### Public Static Functions

```cpp
static pinned_ptr call (naming::address_type lva)
```

```cpp
namespace hpx
```  

```cpp
namespace components
```  

```cpp
template<typename ServerComponent>
struct stub_base
```

### Public Types

```cpp
template<>
using server_component_type = ServerComponent
```

### Public Static Functions

```cpp
static components::component_type get_component_type ()
```

```cpp
namespace hpx
```  

```cpp
namespace components
```  

```cpp
template<typename Component>
class abstract_fixed_component_base : private hpx::traits::detail::fixed_component_tag
```
**Public Types**

```cpp
template<>
using wrapping_type = fixed_component<Component>

template<>
using this_component_type = Component

template<>
using base_type_holder = Component
```

**Public Functions**

```cpp
virtual ~abstract_fixed_component_base()
```

**Public Static Functions**

```cpp
static constexpr component_type get_component_type()
static constexpr void set_component_type(component_type t)
```

```cpp
namespace hpx
```  
```cpp
namespace components
```  
```cpp
#include <abstract_migration_support.hpp>

This hook has to be inserted into the derivation chain of any abstract_component_base for it to support migration.

**Public Types**

```cpp
template<typename BaseComponent, typename Mutex = lcos::local::spinlock>
struct abstract_base_migration_support : public BaseComponent
```

```cpp
#include <abstract_migration_support.hpp> This hook has to be inserted into the derivation chain of any abstract_component_base for it to support migration.
```

**Public Types**

```cpp
template<>
using decorates_action = void
```

**Public Functions**

```cpp
virtual ~abstract_base_migration_support()
```

```cpp
virtual void pin() = 0
```

```cpp
virtual bool unpin() = 0
```

```cpp
virtual std::uint32_t pin_count() const = 0
```

```cpp
virtual void mark_as_migrated() = 0
```

```cpp
virtual hpx::future<void> mark_as_migrated(hpx::id_type const &to_migrate) = 0
```

```cpp
virtual void on_migrated() = 0
```
Public Static Functions

template<typename F>
static threads::thread_function_type decorate_action(naming::address_type lva, F &&f)

Protected Functions

threads::thread_result_type thread_function(threads::thread_function_type &&f, components::pinned_ptr, threads::thread_restart_state state)

Private Types

template<>
using base_type = BaseComponent

template<>
using this_component_type = typename base_type::this_component_type

#include <abstractMigration_support.hpp> This hook has to be inserted into the derivation chain of any component for it to support migration.

Public Types

template<>
using base_type = migration_support<Derived>

template<>
using abstract_base_type = Base

template<>
using wrapping_type = typename base_type::wrapping_type

template<>
using wrapped_type = typename base_type::wrapped_type

template<>
using type_holder = Derived

template<>
using base_type_holder = Base
Public Functions

template<typename ...Ts>
abstract_migration_support (Ts&&... ts)
~abstract_migration_support ()
constexpr void finalize ()

hpx::future<void> mark_as_migrated (hpx::id_type const &to_migrate)

void mark_as_migrated ()

std::uint32_t pin_count () const

void pin ()

bool unpin ()

void on_migrated ()

namespace hpx

namespace traits

template<typename Component, typename Enable>
struct component_heap_type

Public Types

template<>
using type = hpx::components::detail::simple_heap<Component>

Defines

HPX_REGISTER_COMPONENT_HEAP (Component)

namespace hpx

namespace components

Functions

template<typename Component>
Component::heap_type &component_heap ()

namespace hpx

namespace components

namespace server
**Functions**

```cpp
template<typename Component, typename ...Ts>
naming::gid_type create(Ts&&... ts)
    Create a component and forward the passed parameters.
    Create arrays of components using their default constructor.

template<typename Component, typename ...Ts>
naming::gid_type create_migrated(naming::gid_type const &gid, void **p, Ts&&... ts)

template<typename Component, typename ...Ts>
std::vector<naming::gid_type> bulk_create(std::size_t count, Ts&&... ts)
    Create count components and forward the passed parameters.
```

```cpp
namespace hpx

namespace components

namespace server

**Functions**

```cpp
template<typename Component, typename ...Ts>
naming::gid_type construct(Ts&&... ts)
```

```cpp
namespace hpx

namespace components

template<typename BaseComponent, typename Mutex = std::local::spinlock>
struct locking_hook : public BaseComponent
    #include <locking_hook.hpp> This hook can be inserted into the derivation chain of any component allowing to automatically lock all action invocations for any instance of the given component.

**Public Types**

```cpp
template<>
using decorates_action = void
```

**Public Functions**

```cpp
template<typename ...Arg>
locking_hook(Arg&&... arg)
locking_hook(locking_hook const &rhs)
locking_hook(locking_hook &&rhs)
locking_hook &operator=(locking_hook const &rhs)
locking_hook &operator=(locking_hook &&rhs)
```
Public Static Functions

template<typename F>
static threads::thread_function_type decorate_action (naming::address_type lva, F &&f)

Protected Types

template<>
using yield_decorator_type = util::function_nonser<threads::thread_arg_type (threads::thread_result_type)>

Protected Functions

threads::thread_result_type thread_function (threads::thread_function_type f, 
threads::thread_arg_type state)

threads::thread_arg_type yield_function (threads::thread_result_type state)

Private Types

template<>
using mutex_type = Mutex

template<>
using base_type = BaseComponent

template<>
using this_component_type = typename base_type::this_component_type

Private Members

mutex_type mtx_

struct decorate_wrapper

Public Functions

template<typename F, typename Enable = typename std::enable_if<!std::is_same<typename std::decay<F>::type, decorate_wrapper>::value>::type>
decorate_wrapper (F &&f)

template<>
~decorate_wrapper ()

struct undecorate_wrapper

Public Functions

```cpp
template<>
undecorate_wrapper()

template<>
~undecorate_wrapper()
```

Public Members

```cpp
template<>
yield_decorator_type yield_decorator_
```

```cpp
struct init<traits::construct_with_back_ptr>
```

Public Static Functions

```cpp
template<typename Component, typename Managed>
static constexpr void call(Component*, Managed*)
```

```cpp
template<typename Component, typename Managed, typename ...Ts>
static void call_new(Component *&component, Managed *this_, Ts&&... vs)
```

```cpp
struct init<traits::construct_without_back_ptr>
```

Public Static Functions

```cpp
template<typename Component, typename Managed>
static void call(Component *component, Managed *this_)
```

```cpp
template<typename Component, typename Managed, typename ...Ts>
static void call_new(Component *&component, Managed *this_, Ts&&... vs)
```

```cpp
struct destroy_backptr<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```cpp
template<typename BackPtr>
static void call(BackPtr *back_ptr)
```

```cpp
struct destroy_backptr<traits::managed_object_controls_lifetime>
```
Public Static Functions

```cpp
template<typename BackPtr>
static constexpr void call (BackPtr*)
```

```cpp
template<>
struct manage_lifetime<traits::managed_object_is_lifetime_controlled>
```

Public Static Functions

```cpp
template<typename Component>
static constexpr void call (Component*)
```

```cpp
template<typename Component>
static void addr (Component *component)
```

```cpp
template<typename Component>
static void release (Component *component)
```

```cpp
template<>
struct manage_lifetime<traits::managed_object_controls_lifetime>
```

Public Static Functions

```cpp
template<typename Component>
static void call (Component *component)
```

```cpp
template<typename Component>
static constexpr void addr (Component*)
```

```cpp
template<typename Component>
static constexpr void release (Component*)
```

namespace hpx

namespace components

Functions

```cpp
template<typename Component, typename Derived>
void intrusive_ptr_add_ref (managed_component<Component, Derived> *p)
```

```cpp
template<typename Component, typename Derived>
void intrusive_ptr_release (managed_component<Component, Derived> *p)
```

namespace detail_adl_barrier

```cpp
template<>
struct destroy_backptr<traits::managed_object_controls_lifetime>
```
Public Static Functions

template<typename BackPtr>
static constexpr void call (BackPtr*)

template<>
struct destroy_backptr<traits::managed_object_is_lifetime_controlled>

Public Static Functions

template<typename BackPtr>
static void call (BackPtr *back_ptr)

template<>
struct init<traits::construct_with_back_ptr>

Public Static Functions

template<typename Component, typename Managed>
static constexpr void call (Component*, Managed*)

template<typename Component, typename Managed, typename ...Ts>
static void call_new (Component *&component, Managed *this_, Ts&&... vs)

template<>
struct init<traits::construct_without_back_ptr>

Public Static Functions

template<typename Component, typename Managed>
static void call (Component *component, Managed *this_)

template<typename Component, typename Managed, typename ...Ts>
static void call_new (Component *&component, Managed *this_, Ts&&... vs)

template<>
struct manage_lifetime<traits::managed_object_controls_lifetime>

Public Static Functions

template<typename Component>
static void call (Component *component)

template<typename Component>
static constexpr void addr (Component*)

template<typename Component>
static constexpr void release (Component*)

template<>
struct manage_lifetime<traits::managed_object_is_lifetime_controlled>
Public Static Functions

```cpp
template<typename Component>
static constexpr void call (Component*)

template<typename Component>
static void addref (Component *component)

template<typename Component>
static void release (Component *component)
```

```cpp
namespace hpx

namespace components

template<typename BaseComponent, typename Mutex = lcos::local::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

```cpp
template<>
using decorates_action = void
```

Public Functions

```cpp
template<typename ...Arg>
migration_support (Arg&&... arg)

~migration_support()

naming::gid_type get_base_gid (naming::gid_type const &assign_gid = naming::invalid_gid) const

void pin()

bool unpin()

std::uint32_t pin_count() const

void mark_as_migrated()

hpx::future<void> mark_as_migrated (hpx::id_type const &to_migrate)

constexpr void on_migrated()
```

This hook is invoked on the newly created object after the migration has been finished.
Public Static Functions

```cpp
static constexpr bool supports_migration()
```
```cpp
static threads::thread_function_type decorate_action(naming::address_type lva, F &&f)
```
```cpp
static std::pair<bool, components::pinned_ptr> was_object_migrated(hpx::naming::gid_type const &id, naming::address_type lva)
```

Protected Functions

```
threads::thread_result_type thread_function(threads::thread_function_type &&f, components::pinned_ptr, threads::thread_restart_state state)
```

Private Types

```cpp
template<> using mutex_type = Mutex
template<> using base_type = BaseComponent
template<> using this_component_type = typename base_type::this_component_type
```

Private Members

```cpp
mutex_type mtx_
std::uint32_t pin_count_
hpx::lcos::local::promise<void> trigger_migration_
bool was_marked_for_migration_
```

namespace hpx

```cpp
namespace util
```

```cpp
class one_size_heap_list
Subclassed by hpx::components::detail::wrapper_heap_list< Heap >
```
Public Types

using list_type = std::list<std::shared_ptr<util::wrapper_heap_base>>
using iterator = typename list_type::iterator
using const_iterator = typename list_type::const_iterator
using mutex_type = lcos::local::spinlock
using unique_lock_type = std::unique_lock<mutex_type>
using heap_parameters = wrapper_heap_base::heap_parameters

Public Functions

one_size_heap_list()

template<typename Heap>
one_size_heap_list(char const *class_name, heap_parameters parameters, Heap* = nullptr)

template<typename Heap>
one_size_heap_list(std::string const &class_name, heap_parameters parameters, Heap* = nullptr)

~one_size_heap_list()

void *alloc(std::size_t count = 1)

bool reschedule(void *p, std::size_t count)

void free(void *p, std::size_t count = 1)

bool did_alloc(void *p) const

std::string name() const

Public Members

std::shared_ptr<util::wrapper_heap_base>(*create_heap_) (char const*, std::size_t, heap_parameters)

const heap_parameters parameters_

Protected Attributes

mutex_type mtx_

list_type heap_list_
Private Members

`const std::string class_name_`

Private Static Functions

```cpp
template<typename Heap>
static std::shared_ptr<util::wrapper_heap_base> create_heap(char const *name, std::size_t counter, heap_parameters parameters)
```

Defines

`HPX_DEBUG_WRAPPER_HEAP`

```cpp
namespace hpx

namespace util

struct wrapper_heap_base
    Subclassed by hpx::components::detail::wrapper_heap

Public Functions

```cpp
virtual ~wrapper_heap_base() const = 0
virtual bool alloc(void **result, std::size_t count = 1) const = 0
virtual bool did_alloc(void *p) const = 0
virtual void free(void *p, std::size_t count = 1) const = 0
virtual naming::gid_type get gid(util::unique_id_ranges &ids, void *p, components::component_type type) const = 0
virtual std::size_t heap_count() const = 0
virtual std::size_t size() const = 0
virtual std::size_t free_size() const = 0
```

```cpp
struct heap_parameters

```
Public Members

`std::size_t capacity`
`std::size_t element_alignment`
`std::size_t element_size`

namespace hpx

namespace traits

template<typename Action, typename Enable = void>
struct action_decorate_function

Public Static Functions

template<typename F>
static threads::thread_function_type call (naming::address_type lva, F &f)

Public Static Attributes

constexpr bool value = has_decorates_action<Action>::value

namespace traits

template<typename Component, typename Enable = void>
struct component_decorate_function

Public Static Functions

template<typename F>
static threads::thread_function_type call (naming::address_type lva, F &f)

namespace hpx

namespace traits

template<typename Component, typename Enable = void>
struct component_config_data

Public Static Functions

static char const *call ()

namespace hpx

namespace traits

template<typename Component, typename Enable = void>
struct component_pin_support
Public Static Functions

\begin{verbatim}
static constexpr void pin (Component *p)
static constexpr bool unpin (Component *p)
static constexpr std::uint32_t pin_count (Component *p)
\end{verbatim}

namespace hpx

namespace traits

\begin{verbatim}
template<typename Component, typename Enable = void>
struct component_supportsMigration
\end{verbatim}

Public Static Functions

\begin{verbatim}
static constexpr bool call()
\end{verbatim}

namespace hpx

namespace components

TypeDefs

\begin{verbatim}
using component_type = std::int32_t
\end{verbatim}

namespace traits

\begin{verbatim}
template<typename Component, typename Enable = void>
struct component_type_database
\end{verbatim}

Subclassed by hpx::traits::component_type_database< Component const, Enable >

Public Static Functions

\begin{verbatim}
static components::component_type get()
static void set (components::component_type)
\end{verbatim}

Public Static Attributes

\begin{verbatim}
components::component_type value = components::component_type(-1)
\end{verbatim}

namespace hpx

namespace traits

\begin{verbatim}
template<typename Component, typename Enable = void>
struct component_type_is_compatible
\end{verbatim}
Public Static Functions

```
static bool call (naming::address const &addr)
```

namespace hpx

namespace traits

```
template<typename Component>
struct is_fixed_component : public std::integral_constant<bool, std::is_base_of<traits::detail::fixed_component_tag, Component>::value>
```

Subclassed by hpx::traits::is_fixed_component< Component const >

```
template<typename Component>
struct is_managed_component : public std::integral_constant<bool, std::is_base_of<traits::detail::managed_component_tag, Component>::value>
```

Subclassed by hpx::traits::is_managed_component< Component const >

```
template<typename Component>
struct managed_component_ctor_policy<Component, typename util::always_void<typename Component::has_managed_component_base>::type>
```

```
template<typename Component>
struct managed_component_dtor_policy<Component, typename util::always_void<typename Component::has_managed_component_base>::type>
```

Public Types

```
template<>
using type = typename Component::ctor_policy
```

template<typename Component>
```
struct managed_component_ctor_policy<Component, typename util::always_void<typename Component::has_managed_component_base>::type>
```

Public Types

```
template<>
using type = typename Component::dtor_policy
```

namespace hpx

namespace traits

```
template<typename T, typename Enable = void>
struct managed_component_ctor_policy
```

Public Types

```
template<>
using type = construct_without_back_ptr
```

template<typename Component>
```
struct managed_component_ctor_policy<Component, typename util::always_void<typename Component::has_managed_component_base>::type>
```

2.8. API reference 1435
Public Types

```cpp
template<>
using type = typename Component::ctor_policy
```

```cpp
template<typename T, typename Enable = void>
struct managed_component_dtor_policy
```

Public Types

```cpp
template<>
using type = managed_object_controls_lifetime
```

```cpp
template<typename Component>
struct managed_component_dtor_policy<Component, typename util::always_void<typename Component::has_managed_component_base>::type>
```

Public Types

```cpp
template<>
using type = typename Component::dtor_policy
```

compute

The contents of this module can be included with the header `hpx/modules/compute.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/compute.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
```

```cpp
namespace compute
```

Functions

```cpp
template<typename T, typename Allocator>
void swap(vector<T, Allocator> &x, vector<T, Allocator> &y)
    Effects: x.swap(y);
```

```cpp
template<typename T, typename Allocator = std::allocator<T> >
class vector
```
Public Types

template<>
using value_type = T

Member types (FIXME: add reference to std.

template<>
using allocator_type = Allocator

template<>
using access_target = typename alloc_traits::access_target

template<>
using size_type = std::size_t

template<>
using difference_type = std::ptrdiff_t

template<>
using reference = typename alloc_traits::reference

template<>
using const_reference = typename alloc_traits::const_reference

template<>
using pointer = typename alloc_traits::pointer

template<>
using const_pointer = typename alloc_traits::const_pointer

template<>
using iterator = detail::iterator<T, Allocator>

template<>
using const_iterator = detail::iterator<T const, Allocator>

template<>
using reverse_iterator = detail::reverse_iterator<T, Allocator>

template<>
using const_reverse_iterator = detail::const_reverse_iterator<T, Allocator>

Public Functions

vector (Allocator const &alloc = Allocator())

vector (size_type count, T const &value, Allocator const &alloc = Allocator())

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>
vector (InIter first, InIter last, Allocator const &alloc)

vector (vector const &other)

vector (vector const &other, Allocator const &alloc)

vector (vector &&other)

vector (vector &&other, Allocator const &alloc)
vector (std::initializer_list<T> init, Allocator const &alloc)

~vector()

vector &operator=(vector const &other)

vector &operator=(vector &&other)

allocator_type get_allocator() const

Reference operator[] (size_type pos)

const_reference operator[] (size_type pos) const

pointer data()

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 dereferenceable in that case).

const_pointer data() const

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 dereferenceable in that case).

T *device_data() const

Returns a raw pointer corresponding to the address of the data allocated on the device.

std::size_t size() const

std::size_t capacity() const

bool empty() const

Returns: size() == 0.

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.

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.

iterator begin()

iterator end()

const_iterator cbegin() const

const_iterator cend() const

const_iterator begin() const
const_iterator end() const

void swap(vector &other)
    Effects: Exchanges the contents and capacity() of *this with that of x.
    Complexity: Constant time.

default:

void clear()
    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.

Private Types

typedef traits::allocator_traits<Allocator> alloc_traits

Private Members

size_type size_
size_type capacity_
allocator_type alloc_
pointer data_

namespace hpx

namespace compute

namespace host

template<typename T, typename Executor = hpx::parallel::execution::restricted_thread_pool_executor>
struct block_allocator : public hpx::compute::host::detail::policy_allocator<T, hpx::execution::parallel_policy_shim<block_executor<hpx::parallel::execution::restricted_thread_pool_executor>, block_executor<hpx::parallel::execution::restricted_thread_pool_executor>::executor_parameters_type>>
#include <block_allocator.hpp> The block_allocator allocates blocks of memory evenly divided onto the passed vector of targets. This is done by using first touch memory placement.

This allocator can be used to write NUMA aware algorithms:

using allocator_type = hpx::compute::host::block_allocator<int>; using vector_type = hpx::compute::vector<int, allocator_type>;
auto numa_nodes = hpx::compute::host::numa_domains(); std::size_t N = 2048; vector_type v(N, allocator_type(numa_nodes));
Public Types

template<>
using executor_type = block_executor<Executor>

template<>
using executor_parameters_type = typename executor_type::executor_parameters_type

template<>
using policy_type = hpx::execution::parallel_policy_shim<executor_type, executor_parameters_type>

template<>
using base_type = detail::policy_allocator<T, policy_type>

Public Functions

block_allocator()

block_allocator (target_type const &targets)

block_allocator (target_type &&targets)

target_type const &target () const

template<typename Executor>
struct executor_execution_category<compute::host::block_executor<Executor>>

Public Types

typedef hpx::execution::parallel_execution_tag type

namespace hpx

namespace compute

namespace host

# 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

template<>
using executor_parameters_type = hpx::execution::static_chunk_size

Public Functions

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 = { })

block_executor (std::vector<host::target> &&targets)

block_executor (block_executor const &other)

block_executor (block_executor &&other)

block_executor & operator= (block_executor const &other)

block_executor & operator= (block_executor &&other)

template< typename F, typename ...Ts>
void post (F &&f, Ts&&... ts)

template< typename F, typename ...Ts>
hpx::future< typename hpx::util::detail::invoke_deferred_result<F, Ts...>::type> async_execute (F &&f, Ts&&... ts)

template< typename F, typename ...Ts>
hpx::util::detail::invoke_deferred_result<F, Ts...>::type sync_execute (F & &f, Ts&&... ts)

template< typename F, typename Shape, typename ...Ts>
std::vector<hpx::future< typename parallel::execution::detail::bulk_function_result<F, Shape, Ts...>::type>> bulk_async_execute (F &&f, Shape const &shape, Ts&&... ts)

std::vector<host::target> const &targets () const
Private Functions

void init_executors()

Private Members

std::vector<host::target> targets_
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>
struct executor_execution_category<compute::host::block_executor<Executor>>

Public Types

typedef hpx::execution::parallel_execution_tag type

namespace hpx

namespace compute

namespace host

Functions

std::vector<target> get_local_targets()

namespace hpx

namespace parallel

namespace util

template<typename T, typename Executors>
class numa_allocator
Public Types

typedef T value_type

typedef value_type *pointer

typedef value_type const *const_pointer

typedef value_type &reference

typedef value_type const &const_reference

typedef std::size_t size_type

typedef std::ptrdiff_t difference_type

Public Functions

numa_allocator (Executors const &executors, hpx::threads::topology &topo)

numa_allocator (numa_allocator const &rhs)

template<typename U>

numa_allocator (numa_allocator<U, Executors> const &rhs)

pointer address (reference r)

const_pointer address (const_reference r)

pointer allocate (size_type cnt, const void* = nullptr)

void deallocate (pointer p, size_type cnt)

size_type max_size () const

void construct (pointer p, const T&t)

void destroy (pointer p)

Private Types

typedef Executors::value_type executor_type

Private Members

Executors const &executors_

hpx::threads::topology &topo_
Friends

friend hpx::parallel::util::numa_allocator

bool operator==(numa_allocator const&, numa_allocator const&)

bool operator!=(numa_allocator const&, numa_allocator const&)

template<typename U>
struct rebind

Public Types

template<>
typedef numa_allocator<U, Executors> other

Defines

typedef numa_allocator<U, Executors> other

namespace hpx

Functions

static hpx::debug::enable_print<NUMA_BINDING_ALLOCATOR_DEBUG> hpx::nba_deb("NUM_B_A")

namespace compute

namespace host

Typedefs

template<typename T>
using numa_binding_helper_ptr = std::shared_ptr<numa_binding_helper<T>>

template<typename T>
struct numa_binding_allocator

#include <numa_binding_allocator.hpp> The numa_binding_allocator allocates memory using a policy based on hwloc flags for memory binding. This allocator can be used to request data that is bound to one or more numa domains via the bitmap mask supplied

Public Types

typedef T value_type

typedef T* pointer

typedef T& reference

typedef T const&const_reference

typedef std::size_t size_type

typedef std::ptrdiff_t difference_type
template<>

using numa_binding_helper_ptr = std::shared_ptr<numa_binding_helper<T>>

Public Functions

numa_binding_allocator()

numa_binding_allocator(threads::hpx_hwloc_membind_policy policy, unsigned int flags)

numa_binding_allocator(numa_binding_helper_ptr bind_func, threads::hpx_hwloc_membind_policy policy, unsigned int flags)

numa_binding_allocator(numa_binding_allocator const &rhs)

template<typename U>

numa_binding_allocator(numa_binding_allocator<U> const &rhs)

numa_binding_allocator(numa_binding_allocator &&rhs)

numa_binding_allocator &operator=(numa_binding_allocator const &rhs)

numa_binding_allocator &operator=(numa_binding_allocator &&rhs)

pointer address(reference x) const

const_pointer address(const_reference x) const

pointer allocate(size_type n)

void deallocate(pointer p, size_type n)

size_type max_size() const

template<class U, class ...A>

void construct(U *const p, A&&... args)

template<class U>

void destroy(U *const p)

int get_numa_domain(void *page)

std::string get_page_numa_domains(void *addr, std::size_t len) const

void initialize_pages(pointer p, size_t n) const

std::string display_binding(pointer p, numa_binding_helper_ptr helper)

template<typename Binder>

std::shared_ptr<Binder> get_binding_helper_cast() const
Public Members

```cpp
const typedef T* hpx::compute::host::numa_binding_allocator::const_pointer
std::shared_ptr<numa_binding_helper<T>> binding_helper_
threads::hpx_hwloc_membind_policy policy_
unsigned int flags_
```

Protected Functions

```cpp
std::vector<threads::hwloc_bitmap_ptr> create_nodesets (threads::hwloc_bitmap_ptr bitmap) const
void touch_pages (pointer p, size_t n, numa_binding_helper_ptr helper, size_type numa_domain, std::vector<threads::hwloc_bitmap_ptr> const &nodesets) const
void bind_pages (pointer p, size_t n, numa_binding_helper_ptr helper, size_type numa_domain, std::vector<threads::hwloc_bitmap_ptr> const &nodesets) const
```

Private Members

```cpp
std::mutex init_mutex
```

template<typename U>
struct rebind

Public Types

```cpp
template<>
typedef numa_binding_allocator<U> other
```

template<typename T>
struct numa_binding_helper

Public Functions

```cpp
virtual std::size_t operator() (const T*const, const T*const, std::size_t const, std::size_t const) const
virtual ~numa_binding_helper ()
virtual const std::string &pool_name() const
virtual std::size_t memory_bytes() const
virtual std::size_t array_rank() const
virtual std::size_t array_size (std::size_t) const
virtual std::size_t memory_step (std::size_t) const
virtual std::size_t display_step (std::size_t) const
```

1446 Chapter 2. What’s so special about HPX?
virtual std::string description() const

**Public Members**

std::string pool_name_ = "default"

namespace parallel

namespace execution

template<>
struct pool_numa_hint<numa_binding_allocator_tag>

**Public Functions**

int operator()(int const &domain) const

namespace hpx

namespace compute

namespace host

**Functions**

std::vector<target> numa_domains()

namespace hpx

namespace compute

namespace host

struct target

**Public Functions**

target()

target(hpx::threads::mask_type mask)

native_handle_type &native_handle()

native_handle_type const &native_handle() const

std::pair<std::size_t, std::size_t> num_pus() const

void synchronize() const
Public Static Functions

static std::vector<target> get_local_targets() const

Private Functions

void serialize (serialization::input_archive &ar, const unsigned int)
void serialize (serialization::output_archive &ar, const unsigned int)

Private Members

native_handle_type handle_

Friends

friend hpx::compute::host::hpx::serialization::access
bool operator==(target const &lhs, target const &rhs)

struct native_handle_type

Public Functions

native_handle_type() const
native_handle_type(hpx::threads::mask_type mask) const
hpx::threads::mask_type &get_device() const
hpx::threads::mask_type const &get_device() const

Private Members

hpx::threads::mask_type mask_

Friends

friend hpx::compute::host::target

template<>
struct access_target<host::target>
Public Types

typedef host::target target_type

Public Static Functions

template<typename T>
static T const & read(target_type const &, T const *)
template<typename T>
static void write(target_type const &, T * dst, T const * src)

namespace hpx

namespace compute

namespace traits

template<>
struct access_target<host::target>

Public Types

typedef host::target target_type

Public Static Functions

template<typename T>
static T const & read(target_type const &, T const *)
template<typename T>
static void write(target_type const &, T * dst, T const * src)

namespace hpx

namespace compute

namespace traits

template<>
struct access_target<host::target>

Public Types

typedef host::target target_type

Public Static Functions

template<typename T>
static T const & read(target_type const &, T const *)
template<typename T>
static void write(target_type const &, T * dst, T const * src)

namespace hpx

namespace compute

namespace traits

template<>
struct access_target<host::target>

Public Types

typedef host::target target_type

Public Static Functions

template<typename T>
static T const & read(target_type const &, T const *)
template<typename T>
static void write(target_type const &, T * dst, T const * src)

namespace hpx

namespace compute

namespace traits

template<>
struct access_target<host::target>

Public Types

typedef host::target target_type

Public Static Functions

template<typename T>
static T const & read(target_type const &, T const *)
template<typename T>
static void write(target_type const &, T * dst, T const * src)

namespace hpx

namespace compute

namespace traits

template<>
struct access_target<host::target>
Public Types

typedef std::vector<host::target> target_type

Public Static Functions

template<typename T>
static T const &read (target_type const &, T const *t)
template<typename T>
static void write (target_type const &, T *dst, T const *src)

namespace hpx

namespace serialization

Functions

template<typename T, typename Allocator>
void serialize (input_archive &ar, compute::vector<T, Allocator> &v, unsigned)
template<typename T, typename Allocator>
void serialize (output_archive &ar, compute::vector<T, Allocator> const &v, unsigned)

namespace hpx

namespace compute

namespace traits

template<typename Allocator>
struct allocator_traits

Public Types

template<>
using reference = typename detail::get_reference_type<Allocator>::type
template<>
using const_reference = typename detail::get_const_reference_type<Allocator>::type
template<>
using access_target = typename detail::get_target_traits<Allocator>::type
template<>
using target_type = typename access_target::target_type
Public Static Functions

```cpp
static auto target (Allocator const & alloc)
```

```cpp
template<typename ...Ts>
static void bulk_construct (Allocator & alloc, pointer p, size_type count, Ts&& ... vs) 
```

```cpp
static void bulk_destroy (Allocator & alloc, pointer p, size_type count)
```

Private Types

```cpp
template<>
using base_type = std::allocator_traits<Allocator>
```

### compute_cuda

The contents of this module can be included with the header `hpx/modules/compute_cuda.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we **strongly** suggest only including the module header `hpx/modules/compute_cuda.hpp`, not the particular header in which the functionality you would like to use is defined. See **Public API** for a list of names that are part of the public **HPX** API.

### distribution_policies

The contents of this module can be included with the header `hpx/modules/distribution_policies.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we **strongly** suggest only including the module header `hpx/modules/distribution_policies.hpp`, not the particular header in which the functionality you would like to use is defined. See **Public API** for a list of names that are part of the public **HPX** API.

```cpp
namespace hpx
```

```cpp
namespace components
```

### Variables

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE char const* const hpx::components::default_binpacking_counter_name= "/runtime{locality/total}/count/component@
```

```cpp
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.

```cpp
```n
tstruct 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

`binpacking_distribution_policy()`
Default-construct a new instance of a `binpacking_distribution_policy`. This policy will represent one locality (the local locality).

`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).

`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).

`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>  
hpx::future<hpx::id_type> create (Ts&&... vs) const  
Create one object on one of the localities associated by this policy instance
```

**Return** A future holding the global address which represents the newly created object

**Parameters**
- `vs`: [in] The arguments which will be forwarded to the constructor of the new object.

```
template<typename Component, typename ...Ts>  
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
```

**Return** A future holding the list of global addresses which represent the newly created objects
Parameters
- `count`: [in] The number of objects to create
- `vs`: [in] The arguments which will be forwarded to the constructors of the new objects.

```
std::string const &get_counter_name () const
Returns the name of the performance counter associated with this policy instance.
```

```
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

```
namespace hpx
namespace components
```

Variables
```
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

```
colocating_distribution_policy ()
Default-construct a new instance of a colocating_distribution_policy. This policy will represent the local locality.
```

```
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 collocated on

```
template<typename Client, typename Stub>
colocating_distribution_policy operator () (client_base<Client, Stub> 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 collocated on

```
template<typename Component, typename ... Ts>
```
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

**Return** A future holding the global address which represents the newly created object

**Parameters**

- `vs`: [in] The arguments which will be forwarded to the constructor of the new object.

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

**Return** A future holding the list of global addresses which represent the newly created objects

**Parameters**

- `count`: [in] The number of objects to create
- `vs`: [in] The arguments which will be forwarded to the constructors of the new objects.

This function is part of the invocation policy implemented by this class

**Parameters**

- `c`: The continuation to be applied to the result.
- `priority`: The thread priority associated with the operation.
- `vs`: [in] The arguments which will be forwarded to the execution of the continuation.

Returns the number of associated localities for this distribution policy

**Note** This function is part of the creation policy implemented by this class

Returns the locality which is anticipated to be used for the next async operation

This function is part of the invocation policy implemented by this class
Public Types

template<>
using type = hpx::future<

namespace hpx

Variables

const container_distribution_policy container_layout = {}

struct container_distribution_policy : public default_distribution_policy

Public Functions

container_distribution_policy() const
container_distribution_policy operator() (std::size_t num_partitions) const
container_distribution_policy operator() (hpx::id_type const &locality) const
container_distribution_policy operator() (std::vector<hpx::id_type> const &localities) const
container_distribution_policy operator() (std::vector<hpx::id_type> &localities) const

std::size_t get_num_partitions() const
std::vector<hpx::id_type> get_localities() const

Private Functions

template<typename Archive>
void serialize(Archive &ar, const unsigned int)

container_distribution_policy (std::size_t num_partitions, std::vector<hpx::id_type> const &localities)

container_distribution_policy (std::size_t num_partitions, std::vector<hpx::id_type> &localities)

container_distribution_policy (hpx::id_type const &locality)
Private Members

`std::size_t num_partitions_`

Friends

```cpp
friend hpx::hpx::serialization::access
```
Parameters

- \texttt{\textit{vs}}: [in] The arguments which will be forwarded to the constructor of the new object.

```cpp
template<typename Component, typename ...Ts>
hpx::future<std::vector<bulk_locality_result>> \texttt{bulk\_create} (std::size_t \textit{count}, Ts&&... \textit{vs}) const
```

Create multiple objects on the localities associated by this policy instance

\textbf{Note} This function is part of the placement policy implemented by this class

\textbf{Return} A future holding the list of global addresses which represent the newly created objects

Parameters

- \textit{count}: [in] The number of objects to create
- \textit{vs}: [in] The arguments which will be forwarded to the constructors of the new objects.

```cpp
template<typename Action, typename ...Ts>
async_result<Action>::type \texttt{async} (launch policy, Ts&&... \textit{vs}) const
```

```cpp
template<typename Action, typename Callback, typename ...Ts>
async_result<Action>::type \texttt{async\_cb} (launch policy, Callback &&\textit{cb}, Ts&&... \textit{vs}) const
```

\textbf{Note} This function is part of the invocation policy implemented by this class

```cpp
template<typename Action, typename Continuation, typename ...Ts>
bool \texttt{apply} (Continuation &&\textit{c}, threads::thread\_priority \textit{priority}, Ts&&... \textit{vs}) const
```

\textbf{Note} This function is part of the invocation policy implemented by this class

```cpp
template<typename Action, typename ...Ts>
bool \texttt{apply} (threads::thread\_priority \textit{priority}, Ts&&... \textit{vs}) const
```

```cpp
template<typename Action, typename Continuation, typename Callback, typename ...Ts>
bool \texttt{apply\_cb} (Continuation &&\textit{c}, threads::thread\_priority \textit{priority}, Callback &&\textit{cb}, Ts&&... \textit{vs}) const
```

\textbf{Note} This function is part of the invocation policy implemented by this class

```cpp
template<typename Action, typename Callback, typename ...Ts>
bool \texttt{apply\_cb} (threads::thread\_priority \textit{priority}, Callback &&\textit{cb}, Ts&&... \textit{vs}) const
```

```cpp
std::size_t \texttt{get\_num\_localities} () const
```

\textbf{Note} This function is part of the creation policy implemented by this class

```cpp
hpx::id\_type \texttt{get\_next\_target} () const
```

Returns the locality which is anticipated to be used for the next async operation

```cpp
template<typename Action>
struct async\_result
```

\textbf{Note} This function is part of the invocation policy implemented by this class

2.8. API reference 1457
Public Types

```cpp
template<>
using type = hpx::future<typename traits::promise_local_result<typename hpx::traits::extract_action<Action>::remote_result_type>::type>
```

```cpp
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

```cpp
unwrapping_result_policy (id_type const &id)
```

```cpp
unwrapping_result_policy (client_base<Client, Stub> const &client)
```

```cpp
async_result<Action>::type async (launch policy, Ts&&... vs) const
```

```cpp
async_result<Action>::type async (launch::sync_policy, Ts&&... vs) const
```

```cpp
async_result<Action>::type async_cb (launch policy, Callback &&cb, Ts&&... vs) const
```

```cpp
async_result<Action>::type async_cb (launch::sync_policy, Callback &&cb, Ts&&... vs) const
```

```cpp
bool apply (<Continuation &&c, threads::thread_priority priority, Ts&&... vs) const
```

```cpp
bool apply (<Continuation &&c, threads::thread_priority priority, Ts&&... vs) const
```

```cpp
hpx::id_type const &get_next_target () const
```

```cpp
struct async_result
```
Public Types

template<>
using type = typename traits::promise_local_result::type

executors_distributed

The contents of this module can be included with the header hpx/modules/executors_distributed.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/executors_distributed.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace parallel

namespace execution

Functions

template<typename DistPolicy>
distribution_policy_executor<typename std::decay<DistPolicy>::type> make_distribution_policy_executor(DistPolicy &&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 which 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<DistPolicy>::value must evaluate to true.
Public Functions

template<typename DistPolicy_, typename Enable = typename std::enable_if<!std::is_same<distribution_policy_executor, typename std::decay<DistPolicy_>::type>::value>::type>
distribution_policy_executor (DistPolicy_&& policy)

Create a new distribution_policy executor from the given distribution policy

Parameters

• policy: The distribution_policy to create an executor from

Private Members

DistPolicy policy_

include

The contents of this module can be included with the header hpx/modules/include.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/include.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

init_runtime

The contents of this module can be included with the header hpx/modules/init_runtime.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/init_runtime.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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.

Parameters
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 additional local wait time before proceeding.

**Parameters**

- **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.

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.

**Return** This function will always return zero.

**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.

Using this function is an alternative to \(hpx::disconnect\), these functions do not need to be called both.

```cpp
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.

**Return** This function will always return zero.

**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.

Using this function is an alternative to \(hpx::disconnect\), these functions do not need to be called both.

```cpp
HPX_NORETURN void hpx::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`.

```cpp
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.

**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.

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.

**Parameters**

- `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.

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.

**Return** This function will always return zero.

**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.

```cpp
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.
Return This function will always return zero.

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.

```cpp
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.

Return The function returns the value, which has been returned from the user supplied main HPX function (usually hpx_main).

namespace hpx

Functions

```cpp
int init (std::function<int (int argc, char **argv)> 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.

Return The function returns the value, which has been returned from the user supplied f.

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)
int init (std::function<int (int argc, char **)>& f)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.

Return The function returns the value, which has been returned from the user supplied f.

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)

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.

Return The function returns the value, which has been returned from the user supplied f.

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)
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`.

Return The function returns the value, which has been returned from the user supplied `f`.

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`)

```cpp
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 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).

Return The function returns the value, which has been returned from `hpx_main` (or 0 when executed in worker mode).

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`)

```cpp
namespace hpx
```

```cpp
struct init_params
```

#include `<hpx_init_params.hpp>` Parameters used to initialize the HPX runtime through `hpx::init` and `hpx::start`.

2.8. API reference 1465
Public Members

std::reference_wrapper<hpx::program_options::options_description const> desc_cmdline = default_desc
default_desc
std::vector<std::string> cfg

startup_function_type startup
shutdown_function_type shutdown

hpx::runtime_mode mode = hpx::runtime_mode::default_
hpx::resource::partitioner_mode rp_mode = hpx::resource::mode_default
hpx::resource::rp_callback_type rp_callback

namespace hpx

Functions

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 an 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.

Return The function returns true if command line processing succeeded and the runtime system was started successfully. It will return false otherwise.

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 parametermode.

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)
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.

Return The function returns \textit{true} if command line processing succeeded and the runtime system was started successfully. It will return \textit{false} otherwise.

Note If the parameter \textit{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 \textit{argc/argv}. Otherwise it will be executed as specified by the parameter \textit{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 \texttt{nullptr} the HPX runtime environment will be started without invoking \( f \).
- \( \textit{argc} \): [in] The number of command line arguments passed in \textit{argv}. This is usually the unchanged value as passed by the operating system (to \texttt{main()}).
- \( \textit{argv} \): [in] The command line arguments for this application, usually that is the value as passed by the operating system (to \texttt{main()}).
- \( \textit{params} \): [in] The parameters to the \texttt{hpx::start} function (See documentation of \texttt{hpx::init_params})

```cpp
bool start (int \textit{argc}, char **\textit{argv}, init_params const &\textit{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.

Return The function returns \textit{true} if command line processing succeeded and the runtime system was started successfully. It will return \textit{false} otherwise.

Note If the parameter \textit{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 \textit{argc/argv}. Otherwise it will be executed as specified by the parameter \textit{mode}.

Parameters

- \( \textit{argc} \): [in] The number of command line arguments passed in \textit{argv}. This is usually the unchanged value as passed by the operating system (to \texttt{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`)

```cpp
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 an 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.

**Return** The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

**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 `mode` parameter.

**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`)

```cpp
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.

**Return** The function returns `true` if command line processing succeeded and the runtime system was started successfully. It will return `false` otherwise.

**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`)

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.

Return This function will always return zero.

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.

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.

Return This function will always return zero.

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.

namespace hpx_startup
Functions

```cpp
std::vector<std::string> user_main_config (std::vector<std::string> const &cfg)
```

Variables

```cpp
std::vector<std::string> (*user_main_config_function) (std::vector<std::string> const &)
```

lcos_distributed

The contents of this module can be included with the header `hpx/modules/lcos_distributed.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/lcos_distributed.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx

namespace lcos

template<typename T>
class channel

Public Types

```cpp
template<>
using value_type = T
```

Public Functions

```cpp
channel ()
channel (naming::id_type const &loc)
channel (hpx::future<naming::id_type> &&&id)
channel (hpx::shared_future<naming::id_type> &&&id)
channel (hpx::shared_future<naming::id_type> const &id)

hpx::future<T> get (launch::async_policy, std::size_t generation = default_generation) const
hpx::future<T> get (std::size_t generation = default_generation) const
T get (launch::sync_policy, std::size_t generation = default_generation, hpx::error_code &ec = hpx::throws) const
T get (launch::sync_policy, hpx::error_code &ec, std::size_t generation = default_generation) const

template<typesizeof U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value, bool> set (launch::apply_policy, U val, std::size_t generation = default_generation)
```
template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value, hpx::future<void>> set (launch::async_policy, U val, std::size_t generation = default_generation)

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value> set (launch::sync_policy, U val, std::size_t generation = default_generation)

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value && !traits::is_launch_policy<U>::value> set (U val, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value, bool> set (launch::apply_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value, hpx::future<void>> set (launch::async_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<std::is_void<U>::value> set (launch::sync_policy, std::size_t generation = default_generation)

void close (launch::apply_policy, bool force_delete_entries = false)

hpx::future<std::size_t> close (launch::async_policy, bool force_delete_entries = false)

std::size_t close (launch::sync_policy, bool force_delete_entries = false)

std::size_t close (bool force_delete_entries = false)

channel_iterator<T, channel<T>> begin () const

channel_iterator<T, channel<T>> end () const

channel_iterator<T, channel<T>> rbegin () const

channel_iterator<T, channel<T>> rend () const
Private Types

```cpp
template<>
using base_type = components::client_base<channel<T>, lcos::server::channel<T>>
```

Private Static Attributes

```cpp
constexpr std::size_t default_generation = std::size_t(-1)
```

```cpp
template<typename T, typename Channel>
class channel_iterator : public hpx::util::iterator_facade<channel_iterator<T, Channel>, T const, std::input_iterator_tag>
```

Public Functions

```cpp
channel_iterator()
channel_iterator(Channel const &c)
```

Private Types

```cpp
template<>
using base_type = hpx::util::iterator_facade<channel_iterator<void, Channel>, T const, std::input_iterator_tag>
```

Private Functions

```cpp
std::pair<T, bool> get_checked() const
bool equal (channel_iterator const &rhs) const
void increment ()
base_type::reference dereference () const
```

Private Members

```cpp
Channel const *channel_
std::pair<T, bool> data_
```

Friends

```cpp
friend hpx::lcos::hpx::util::iterator_core_access
```

```cpp
template<typename Channel>
class channel_iterator<void, Channel> : public hpx::util::iterator_facade<channel_iterator<void, Channel>, ...
```
Public Functions

channel_iterator()
channel_iterator(Channel const &c)

Private Types

template<>
using base_type = hpx::util::iterator_facade<channel_iterator<void, Channel>, util::unused_type const, std::input_iterator_tag>

Private Functions

bool get_checked()
bool equal(channel_iterator const &rhs) const
void increment()
base_type::reference dereference() const

Private Members

Channel const *channel_
bool data_

Friends

friend hpx::lcos::hpx::util::iterator_core_access

template<typename T>
class receive_channel

Public Types

template<>
using value_type = T

Public Functions

receive_channel()
receive_channel(channel<T> const &c)
receive_channel(hpx::future<naming::id_type> &&id)
receive_channel(hpx::shared_future<naming::id_type> &&id)
receive_channel(hpx::shared_future<naming::id_type> const &id)

hpx::future<T> get(launch::async_policy, std::size_t generation = default_generation) const
\texttt{hpx::future\langle T \rangle} \textbf{get} (\texttt{std::size\_t generation = default\_generation}) \textbf{const}

\texttt{T get} (\texttt{launch::sync\_policy, std::size\_t generation = default\_generation, hpx::error\_code \& ec = hpx::throws}) \textbf{const}

\texttt{T get} (\texttt{launch::sync\_policy, hpx::error\_code \& ec, std::size\_t generation = default\_generation}) \textbf{const}

\begin{verbatim}
channel_iterator\langle T, channel\langle T \rangle\rangle \textbf{begin} () \textbf{const}
channel_iterator\langle T, channel\langle T \rangle\rangle \textbf{end} () \textbf{const}
channel_iterator\langle T, channel\langle T \rangle\rangle \textbf{rbegin} () \textbf{const}
channel_iterator\langle T, channel\langle T \rangle\rangle \textbf{rend} () \textbf{const}
\end{verbatim}

\textbf{Private Types}

\begin{verbatim}
template<>
\textbf{using} \textbf{base\_type} = \textit{components::client\_base\langle receive\_channel\langle T \rangle, lcos::server::channel\langle T \rangle\rangle}
\end{verbatim}

\textbf{Private Static Attributes}

\begin{verbatim}
constexpr \texttt{std::size\_t default\_generation} = \texttt{std::size\_t(-1)}
\end{verbatim}

\begin{verbatim}
template<typename T>
\textbf{class} \textbf{send\_channel}
\end{verbatim}

\textbf{Public Types}

\begin{verbatim}
template<>
\textbf{using} \textbf{value\_type} = \texttt{T}
\end{verbatim}

\textbf{Public Functions}

\texttt{send\_channel} ()

\texttt{send\_channel} (channel\langle T \rangle \textbf{const} \& c)

\texttt{send\_channel} (hpx::future\langle naming::id\_type \rangle \&\& id)

\texttt{send\_channel} (hpx::shared\_future\langle naming::id\_type \rangle \&\& id)

\texttt{send\_channel} (hpx::shared\_future\langle naming::id\_type \rangle \textbf{const} \& id)

\begin{verbatim}
template<typename U, typename U2 = T>
\texttt{std::enable\_if\_t<!\texttt{std::is\_void\langle U\rangle::value, bool}> set} (launch::apply\_policy, \texttt{U val, std::size\_t generation = default\_generation})
\end{verbatim}

\begin{verbatim}
template<typename U, typename U2 = T>
\texttt{std::enable\_if\_t<!\texttt{std::is\_void\langle U\rangle::value, hpx::future\langle void\rangle}> set} (launch::async\_policy, \texttt{U val, std::size\_t generation = default\_generation})
\end{verbatim}

\begin{verbatim}
template<typename U, typename U2 = T>
\end{verbatim}
std::enable_if_t<!std::is_void<U>::value> set (launch::sync_policy, U val, std::size_t generation = default_generation)

template<typename U, typename U2 = T>
std::enable_if_t<!std::is_void<U2>::value && !traits::is_launch_policy<U>::value> set (U val, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<!std::is_void<U>::value, bool> set (launch::apply_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<!std::is_void<U>::value, hpx::future<void>> set (launch::async_policy, std::size_t generation = default_generation)

template<typename U = T>
std::enable_if_t<!std::is_void<U>::value> set (std::size_t generation = default_generation)

void close (launch::apply_policy, bool force_delete_entries = false)

hpx::future<std::size_t> close (launch::async_policy, bool force_delete_entries = false)

std::size_t close (launch::sync_policy, bool force_delete_entries = false)

std::size_t close (bool force_delete_entries = false)

Private Types

template<>
using base_type = components::client_base<send_channel<T>, lcos::server::channel<T>>

Private Static Attributes

constexpr std::size_t default_generation = std::size_t(-1)

namespace hpx

namespace lcos

template<typename ValueType>
struct object_semaphore : public components::client_base<object_semaphore<ValueType>, lcos::server::object_semaphore<ValueType>, lcos::server::object_semaphore<ValueType> metavalue = false, lcos::server::object_semaphore<ValueType> {...}}
Public Types

template<>  
using server_type = lcos::server::object_semaphore<ValueType>  
template<>  
using base_type = components::client_base<object_semaphore, lcos::server::object_semaphore<ValueType>>

Public Functions

object_semaphore()  
object_semaphore(naming::id_type gid)  
lcos::future<void> signal(launch::async_policy, Value_Type const &val, std::uint64_t count = 1)  
void signal(launch::sync_policy, Value_Type const &val, std::uint64_t count = 1)  
lcos::future<Value_Type> get(launch::async_policy)  
Value_Type get(launch::sync_policy)  
future<void> abort_pending(launch::async_policy, error ec = no_success)  
void abort_pending(launch::sync_policy, error = no_success)  
void wait(launch::async_policy)  
void wait(launch::sync_policy)

Defines

HPX_REGISTER_CHANNEL_DECLARATION(...)  
HPX_REGISTER_CHANNEL_DECLARATION_(...)  
HPX_REGISTER_CHANNEL_DECLARATION_1(type)  
HPX_REGISTER_CHANNEL_DECLARATION_2(type, name)  
HPX_REGISTER_CHANNEL(...)  
HPX_REGISTER_CHANNEL_(...)  
HPX_REGISTER_CHANNEL_1(type)  
HPX_REGISTER_CHANNEL_2(type, name)  
namespace hpx

    namespace lcos

    namespace server

    template<typename T, typename Remote_Type>
    class channel
Public Types

```cpp
template<>
using base_type_holder = lcos::base_lco_with_value<T, RemoteType, traits::detail::component_tag>
```

```cpp
template<>
using wrapping_type = typename base_type::wrapping_type
```

Public Functions

```cpp
channel()  // channel
void set_value (RemoteType &&result)  // set_value
void set_exception (std::exception_ptr const&)  // set_exception
result_type get_value ()  // get_value
result_type get_value (error_code &ec)  // get_value
hpx::future<T> get_generation (std::size_t generation)  // get_generation
HPX_DEFINE_COMPONENT_DIRECT_ACTION (channel, get_generation)  // HPX_DEFINE_COMPONENT_DIRECT_ACTION
void set_generation (RemoteType &&value, std::size_t generation)  // set_generation
HPX_DEFINE_COMPONENT_DIRECT_ACTION (channel, set_generation)  // HPX_DEFINE_COMPONENT_DIRECT_ACTION
std::size_t close (bool force_delete_entries)  // close
HPX_DEFINE_COMPONENT_ACTION (channel, close)  // HPX_DEFINE_COMPONENT_ACTION
```

Public Static Functions

```cpp
static components::component_type get_component_type ()  // get_component_type
static void set_component_type (components::component_type type)  // set_component_type
```

Private Types

```cpp
template<>
using base_type = components::component_base<channel>
```

```cpp
template<>
using result_type = std::conditional_t<std::is_void<T>::value, util::unused_type, T>  // result_type
```
Private Members

```cpp
namespace hpx

namespace lcos

namespace server

template<typename ValueType>
struct object_semaphore : public components::managed_component_base<object_semaphore<ValueType>>
```

Public Types

```cpp
template<>
using base_type = components::managed_component_base<object_semaphore>

template<>
using mutex_type = hpx::lcos::local::spinlock

template<>
using slist_option_type = boost::intrusive::member_hook<queue_thread_entry, typename queue_thread_entry::hook_type, &queue_thread_entry::slist_hook_>

template<>
using thread_queue_type = boost::intrusive::slist<queue_thread_entry, slist_option_type, boost::intrusive::cache_last<true>, boost::intrusive::constant_time_size<false> >

template<>
using value_slist_option_type = boost::intrusive::member_hook<queue_value_entry, typename queue_value_entry::hook_type, &queue_value_entry::slist_hook_>

template<>
using value_queue_type = boost::intrusive::slist<queue_value_entry, value_slist_option_type, boost::intrusive::cache_last<true>, boost::intrusive::constant_time_size<false> >
```

Public Functions

```cpp
object_semaphore()  
~object_semaphore()  
void signal (ValueType const &val, std::uint64_t count)  
void get (naming::id_type const &lco)  
void abort_pending (error ec)  
void wait ()  
HPX_DEFINE_COMPONENT_ACTION (object_semaphore, signal, signal_action)  
HPX_DEFINE_COMPONENT_ACTION (object_semaphore, get, get_action)  
HPX_DEFINE_COMPONENT_ACTION (object_semaphore, abort_pending, abort_pending_action)  
HPX_DEFINE_COMPONENT_ACTION (object_semaphore, wait, wait_action)
```
Private Functions

void resume (std::unique_lock<mutex_type> &l)

Private Members

value_queue_type value_queue_
thread_queue_type thread_queue_
mutex_type mtx_

struct queue_thread_entry

Public Types

template<>
typedef boost::intrusive::slist_member_hook<boost::intrusive::link_mode<boost::intrusive::normal_link>> hook_type

Public Functions

template<>
queue_thread_entry (naming::id_type const &id)

Public Members

template<>
naming::id_type id_

template<>
hook_type slist_hook_

struct queue_value_entry

Public Types

template<>
typedef boost::intrusive::slist_member_hook<boost::intrusive::link_mode<boost::intrusive::normal_link>> hook_type

Public Functions

template<>
queue_value_entry (ValueType const &val, std::uint64_t count)
Public Members

template<>
ValueType val_

template<>
std::uint64_t count_

template<>

hook_type slist_hook_

naming

The contents of this module can be included with the header hpx/modules/naming.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/naming.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

namespace naming

Functions

void decrement_refcnt (gid_type const &gid)

void save (serialization::output_archive &ar, id_type const&, unsigned int)

void load (serialization::input_archive &ar, id_type&, unsigned int)

naming_base

The contents of this module can be included with the header hpx/modules/naming_base.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/naming_base.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

Defines

HPX_ADDRESS_VERSION

namespace hpx

namespace naming

struct address
Public Types

using component_type = naming::component_type
using address_type = naming::address_type

Public Functions

constexpr address ()
constexpr address (gid_type const &l, component_type t = component_invalid)
address (gid_type const &l, component_type t, void *lva)
constexpr address (gid_type const &l, component_type t, address_type a)
address (void *lva, component_type t = component_invalid)
constexpr address (address_type a)
constexpr operator bool () const

Public Members

gid_type locality_
component_type type_ = component_invalid
address_type address_ = 0

Public Static Attributes

constexpr const component_type component_invalid = -1

Private Functions

template<typername Archive>
void save (Archive &ar, unsigned int version) const

template<typername Archive>
void load (Archive &ar, unsigned int version)

HPX_SERIALIZATION_SPLIT_MEMBER ()

Friends

friend hpx::naming::hpx::serialization::access
        address (local virtual address)

friend constexpr bool operator==(address const &lhs, address const &rhs)
Defines

HPX_GIDTYPE_VERSION
template<> 
struct hash<hpx::naming::gid_type>

Public Functions

std::size_t operator() (::hpx::naming::gid_type const &gid) const

namespace hpx

namespace naming

Functions

gid_type operator+ (gid_type const &lhs, gid_type const &rhs)
gid_type operator- (gid_type const &lhs, gid_type const &rhs)
void save (serialization::output_archive &ar, gid_type const &,
unsigned int)
void load (serialization::input_archive &ar, gid_type&,
unsigned int version)
gid_type get_gid_from_locality_id (std::uint32_t locality_id)
std::uint32_t get_locality_id_from_gid (std::uint64_t msb)
std::uint32_t get_locality_id_from_gid (gid_type const &id)
gid_type get_locality_from_gid (gid_type const &id)
bool is_locality (gid_type const &gid)
std::uint64_t replace_locality_id (std::uint64_t msb,
std::uint32_t locality_id)
gid_type replace_locality_id (gid_type const &gid,
std::uint32_t locality_id)
constexpr bool refers_to_virtual_memory (std::uint64_t msb)
constexpr bool refers_to_virtual_memory (gid_type const &gid)
constexpr bool refers_to_local_lva (gid_type const &gid)
gid_type replace_component_type (gid_type const &gid,
std::uint32_t type)
std::ostream &operator<< (std::ostream &os, gid_type const &id)
Variables

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE const gid_type hpx::naming::invalid_gid = {};
```

```cpp
struct gid_type
#include <gid_type.hpp>
Global identifier for components across the HPX system.
Subclassed by hpx::naming::detail::id_type_impl
```

Public Types

```cpp
using size_type = gid_type
using difference_type = gid_type
using mutex_type = gid_type
```

Public Functions

```cpp
constexpr gid_type();
constexpr gid_type(std::uint64_t lsb_id);
gid_type(std::uint64_t msb_id, std::uint64_t lsb_id);
constexpr gid_type(gid_type const &rhs);
constexpr gid_type(gid_type &&rhs);
~gid_type();
gid_type &operator=(std::uint64_t lsb_id);
gid_type &operator=(gid_type const &rhs);
gid_type &operator=(gid_type &&rhs);
constexpr operator bool() const;
gid_type &operator++();
gid_type operator++(int);
gid_type &operator--();
gid_type operator--(int);
gid_type &operator+=(gid_type const &rhs);
gid_type &operator+=(std::uint64_t rhs);
gid_type &operator-=(gid_type const &rhs);
gid_type &operator-=(std::uint64_t rhs);
constexpr std::uint64_t get_msb() const;
constexpr void set_msb(std::uint64_t msb);
```
constexpr std::uint64_t get_lsb () const
constexpr void set_lsb (std::uint64_t lsb)
void set_lsb (void *lsb)
std::string to_string () const
void lock ()
bool try_lock ()
void unlock ()
mutex_type &get_mutex () const

Public Static Attributes

constexpr std::uint64_t credit_base_mask = 0x1full
constexpr std::uint16_t credit_shift = 24
constexpr std::uint64_t credit_mask = credit_base_mask << credit_shift
constexpr std::uint64_t was_split_mask = 0x80000000ull
constexpr std::uint64_t has_credits_mask = 0x40000000ull
constexpr std::uint64_t is_locked_mask = 0x20000000ull
constexpr std::uint64_t locality_id_mask = 0xffffffff00000000ull
constexpr std::uint16_t locality_id_shift = 32
constexpr std::uint64_t virtual_memory_mask = 0x3ffffull
constexpr std::uint64_t dont_cache_mask = 0x800000ull
constexpr std::uint64_t is_migratable = 0x400000ull
constexpr std::uint64_t dynamically_assigned = 0x1ull
constexpr std::uint64_t component_type_base_mask = 0xfffffull
constexpr std::uint64_t component_type_shift = 1ull
constexpr std::uint64_t component_type_mask = component_type_base_mask << component_type_shift
constexpr std::uint64_t credit_bits_mask = credit_mask | was_split_mask | has_credits_mask
constexpr std::uint64_t internal_bits_mask = credit_bits_mask | is_locked_mask | dont_cache_mask | is_migratable
constexpr std::uint64_t special_bits_mask = locality_id_mask | internal_bits_mask | component_type_mask
**Private Types**

```cpp
using spinlock_pool = util::spinlock_pool<gid_type>
```

**Private Functions**

```cpp
bool acquire_lock()
void relinquish_lock()
constexpr bool is_locked() const
```

**Private Members**

```cpp
std::uint64_t id_msb_ = 0
std::uint64_t id_lsb_ = 0
```

**Friends**

```cpp
gid_type operator+ (gid_type const &lhs, gid_type const &rhs)
gid_type operator+ (gid_type const &lhs, std::uint64_t rhs)
gid_type operator- (gid_type const &lhs, gid_type const &rhs)
gid_type operator- (gid_type const &lhs, std::uint64_t rhs)
gid_type operator& (gid_type const &lhs, std::uint64_t rhs)
bool operator== (gid_type const &lhs, gid_type const &rhs)
bool operator!= (gid_type const &lhs, gid_type const &rhs)
bool operator< (gid_type const &lhs, gid_type const &rhs)
bool operator>= (gid_type const &lhs, gid_type const &rhs)
bool operator<= (gid_type const &lhs, gid_type const &rhs)
bool operator> (gid_type const &lhs, gid_type const &rhs)
std::ostream &operator<< (std::ostream &os, gid_type const &id)
void save (serialization::output_archive &ar, gid_type const&, unsigned int)
void load (serialization::input_archive &ar, gid_type&, unsigned int version)
```

```cpp
namespace std

    template<>
    struct hash<hpx::naming::gid_type>
```
Public Functions

`std::size_t operator()(::hpx::naming::gid_type const &id) const`

```
template<>
struct get_remote_result<naming::id_type, naming::gid_type>
```

Public Static Functions

```
static naming::id_type call(naming::gid_type const &rhs)
```

```
template<>
struct promise_local_result<naming::gid_type>
```

Public Types

```
typedef naming::id_type type
```

```
typedef std::vector<naming::id_type> type
```

namespace hpx

namespace naming

Functions

```
std::ostream &operator<<(std::ostream &os, id_type const &id)
```

```
char const *get_management_type_name(id_type::management_type m)
```

```
id_type get_id_from_locality_id(std::uint32_t locality_id)
```

```
std::uint32_t get_locality_id_from_id(id_type const &id)
```

```
id_type get_locality_from_id(id_type const &id)
```

```
bool is_locality(id_type const &id)
```

```
bool operator!=(id_type const &lhs, id_type const &rhs)
```

```
bool operator<=(id_type const &lhs, id_type const &rhs)
```
bool operator> (id_type const &lhs, id_type const &rhs)
bool operator>=(id_type const &lhs, id_type const &rhs)

**Variables**

const id_type invalid_id = id_type()

**struct id_type**

**Public Types**

class enum management_type

Values:

- unknown_deleter = -1
- unmanaged = 0
  - unmanaged GID
- managed = 1
  - managed GID
- managed_move_credit = 2
  - managed GID which will give up all credits when sent

**Public Functions**

constexpr id_type ()

id_type (std::uint64_t lsb_id, management_type t)

id_type (gid_type const &gid, management_type t)

id_type (std::uint64_t msb_id, std::uint64_t lsb_id, management_type t)

id_type (id_type const &o)

id_type (id_type &&o)

id_type &operator= (id_type const &o)

id_type &operator= (id_type &&o)

gid_type const &get_gid ()

gid_type const &get_gid () const

id_type::management_type get_management_type () const

id_type &operator++ ()

id_type operator++ (int)

operator bool () const

std::uint64_t get_msb () const
void **set_msb**(std::uint64_t msb)

**std::uint64_t get_lsb** () const

void **set_lsb**(std::uint64_t lsb)

void **set_lsb**(void *lsb)

void **make_unmanaged**( () const

**hpx::intrusive_ptr<detail::id_type_impl> &impl** ()

**hpx::intrusive_ptr<detail::id_type_impl> const &impl** () const

**Private Members**

**hpx::intrusive_ptr<detail::id_type_impl> gid**

**Friends**

bool **operator==(** (id_type const &lhs, id_type const &rhs)

bool **operator!=(** (id_type const &lhs, id_type const &rhs)

bool **operator<**( (id_type const &lhs, id_type const &rhs)

bool **operator<=**( (id_type const &lhs, id_type const &rhs)

bool **operator>**( (id_type const &lhs, id_type const &rhs)

bool **operator>=**( (id_type const &lhs, id_type const &rhs)

**std::ostream &operator<<(** (std::ostream &os, id_type const &id)

**namespace traits**

**template<>**

**struct get_remote_result**<naming::id_type, naming::gid_type>

**Public Static Functions**

**static naming::id_type call**( naming::gid_type const &rhs)

**template<>**

**struct get_remote_result**<std::vector<naming::id_type>, std::vector<naming::gid_type>>
Public Static Functions

```cpp
static std::vector<naming::id_type> call(std::vector<naming::gid_type> const &rhs)
```

```cpp
template<>
struct promise_local_result<naming::gid_type>
```

Public Types

```cpp
typedef naming::id_type type
```

```cpp
template<>
struct promise_local_result<std::vector<naming::gid_type>>
```

Public Types

```cpp
typedef std::vector<naming::id_type> type
```

namespace hpx

namespace naming

Typedefs

```cpp
using component_type = std::int32_t
```

```cpp
using address_type = std::uint64_t
```

Variables

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE std::uint32_t hpx::naming::invalid_locality_id = ~static_cast<std::uint32_t>(0)
```

```cpp
HPX_INLINE_CONSTEXPR_VARIABLE std::int32_t hpx::naming::component_invalid = -1
```

namespace hpx

namespace naming

Functions

```cpp
id_type unmanaged(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.

Return 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.

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.

**performance_counters**

The contents of this module can be included with the header `hpx/modules/performance_counters.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we *strongly* suggest only including the module header `hpx/modules/performance_counters.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

```cpp
namespace hpx
namespace performance_counters

Functions

bool action_invocation_counter_discoverer (hpx::actions::detail::invocation_count_registry const &registry, counter_info const &info, counter_path_elements &p, discover_counter_func const &f, discover_counters_mode mode, error_code &ec)
```

```cpp
namespace hpx
namespace agas

Enums

```cpp
enum namespace_action_code

Values:

- invalid_request = 0
- locality_ns_service = 0b1100000
- locality_ns_bulk_service = 0b1100001
- locality_ns_allocate = 0b1100010
- locality_ns_free = 0b1100011
```
HPX Documentation, master

```plaintext
locality_ns_localities = 0b1100100
locality_ns_num_localities = 0b1100101
locality_ns_num_threads = 0b1100110
locality_ns_statistics_counter = 0b1100111
locality_ns_resolve_locality = 0b1101000
primary_ns_service = 0b1000000
primary_ns_bulk_service = 0b1000001
primary_ns_route = 0b1000010
primary_ns_bind_gid = 0b1000011
primary_ns_resolve_gid = 0b1000100
primary_ns_unbind_gid = 0b1000101
primary_ns_increment_credit = 0b1000110
primary_ns_decrement_credit = 0b1000111
primary_ns_allocate = 0b1001000
primary_ns_begin_migration = 0b1001001
primary_ns_end_migration = 0b1001010
primary_ns_statistics_counter = 0b1001011
component_ns_service = 0b0100000
component_ns_bulk_service = 0b0100001
component_ns_bind_prefix = 0b0100010
component_ns_bind_name = 0b0100011
component_ns_resolve_id = 0b0100100
component_ns_unbind_name = 0b0100101
component_ns_iterate_types = 0b0100110
component_ns_get_component_type_name = 0b0100111
component_ns_num_localities = 0b0101000
component_ns_statistics_counter = 0b0101001
symbol_ns_service = 0b0010000
symbol_ns_bulk_service = 0b0010001
symbol_ns_bind = 0b0010010
symbol_ns_resolve = 0b0010011
symbol_ns_unbind = 0b0010100
symbol_ns_iterate_names = 0b0010101
symbol_ns_on_event = 0b0010110
symbol_ns_statistics_counter = 0b0010111
```

2.8. API reference
**Variables**

```cpp
constexpr char const *const performance_counter_baseName = "/agas/"
```

**namespace hpx**

**namespace performance_counters**

```cpp
template<
    typename Derived>

class base_performance_counter
```

**Public Types**

```cpp
typedef Derived type_holder

typedef hpx::performance_counters::server::base_performance_counter base_type_holder
```

**Public Functions**

```cpp
base_performance_counter()  

base_performance_counter(hpx::performance_counters::counter_info const &info)  

void finalize()  
```

**Private Types**

```cpp
typedef hpx::components::component_base<Derived> base_type
```

**namespace hpx**

**namespace agas**

**Functions**

```cpp
void component_namespace_register_counter_types(error_code &ec = throws)
```

**namespace hpx**

**namespace performance_counters**
Functions

```cpp
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.

```cpp
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:

```xml
<objectname>(locality#<locality_id>/total)/<instancename>
```

```cpp
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:

```xml
<objectname>({locality#<locality_id>/pool#<pool_name>/total})/<instancename>
```

```cpp
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:

```xml
<objectname>({locality#0/total})/<instancename>
```

```cpp
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:

```xml
<objectname>({locality#<locality_id>/worker-thread#<threadnum>})/<instancename>
```

```cpp
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:

```xml
<objectname>({locality#<locality_id>/pool#<poolname>/thread#<threadnum>})/<instancename>
```

```cpp
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:

```xml
<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::util::function_nonser<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::util::function_nonser<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&)

namespace hpx

namespace performance_counters

Functions

hpx::future<id_type> create_performance_counter_async (id_type target_id,
    counter_info const &info)

id_type create_performance_counter (id_type target_id, counter_info const &info,
    error_code &ec = throws)

namespace hpx

namespace performance_counters
Functions

bool parse_counter_name (std::string const &name, path_elements &elements)

struct instance_elements

Public Members

instance_name parent_
instance_name child_
instance_name subchild_

struct instance_name

Public Members

std::string name_
std::string index_
bool basename_ = false

struct path_elements

Public Members

std::string object_
instance_elements instance_
std::string counter_
std::string parameters_

namespace hpx

namespace performance_counters

Typedefs

typedef hpx::util::function_nonser<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::util::function_nonser<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::util::function_nonser<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 counter_type

 Values:

counter_text
counter_text shows a variable-length text string. It does not deliver calculated values.

 Formula: None Average: None Type: Text

counter_raw
counter_raw shows the last observed value only. It does not deliver an average.

 Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

counter_monotonically_increasing

counter_average_base
counter_average_base is used as the base data (denominator) in the computation of time or count averages for the counter_average_count and counter_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

counter_average_count
counter_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

counter_aggregating
counter_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)

counter_average_timer
counter_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

counter_elapsed_time
counter_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: \((D_0 - N_0) / F\), 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: \((D_x - N_0) / F\) Type: Difference

counter_histogram

counter_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.

counter_raw_values

counter_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.

counter_text

counter_text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

counter_raw

counter_raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

counter_monotonically_increasing

counter_average_base

counter_average_base is used as the base data (denominator) in the computation of time or count averages for the counter_average_count and counter_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: \(\text{SUM} (N) / x\) Type: Instantaneous

counter_average_count

counter_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: \((N_1 - N_0) / (D_1 - D_0)\), 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: \((N_x - N_0) / (D_x - D_0)\) Type: Average

counter_aggregating

counter_aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: \(F(N_x)\)

counter_average_timer

counter_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: \( \frac{(N_1 - N_0)}{F} \) \( / \) \( \frac{(D_1 - D_0)}{F} \), 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: \( \frac{(N_x - N_0)}{F} \) \( / \) \( \frac{(D_x - D_0)}{F} \) Type: Average

**counter_elapsed_time**

*counter_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: \( \frac{(D_0 - N_0)}{F} \), 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: \( \frac{(D_x - N_0)}{F} \) Type: Difference

**counter_histogram**

*counter_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.

**counter_raw_values**

*counter_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.

**enum counter_status**

Status and error codes used by the functions related to performance counters.

**Values:**

- **status_valid_data**
  
  No error occurred, data is valid.

- **status_new_data**
  
  Data is valid and different from last call.

- **status_invalid_data**
  
  Some error occurred, data is not value.

- **status_already_defined**
  
  The type or instance already has been defined.

- **status_counter_unknown**
  
  The counter instance is unknown.

- **status_counter_type_unknown**
  
  The counter type is unknown.

- **status_generic_error**
  
  A unknown error occurred.

- **status_valid_data**
  
  No error occurred, data is valid.
**status_new_data**
Data is valid and different from last call.

**status_invalid_data**
Some error occurred, data is not value.

**status_already_defined**
The type or instance already has been defined.

**status_counter_unknown**
The counter instance is unknown.

**status_counter_type_unknown**
The counter type is unknown.

**status_generic_error**
A unknown error occurred.

### Functions

```cpp
std::string &ensure_counter_prefix (std::string &name)
```

```cpp
std::string ensure_counter_prefix (std::string const &counter)
```

```cpp
std::string &remove_counter_prefix (std::string &name)
```

```cpp
std::string remove_counter_prefix (std::string const &counter)
```

```cpp
char const *get_counter_type_name (counter_type state)
```

Return the readable name of a given counter type.

```cpp
bool status_is_valid (counter_status s)
```

```cpp
counter_status add_counter_type (counter_info const &info, error_code &ec)
```

```cpp
naming::id_type get_counter (std::string const &name, error_code &ec)
```

```cpp
naming::id_type get_counter (counter_info const &info, error_code &ec)
```

### Variables

```cpp
constexpr const char counter_prefix[] = "/counters"
```

```cpp
constexpr std::size_t counter_prefix_len = (sizeof(counter_prefix) / sizeof(counter_prefix[0])) - 1
```

```cpp
struct counter_info
```

### Public Functions

```cpp
counter_info (counter_type type = counter_raw)
```

```cpp
counter_info (std::string const &name)
```

```cpp
counter_info (counter_type type, std::string const &name, std::string const &help-text = "", std::uint32_t version = HPX_PERFORMANCE_COUNTER_V1, std::string const &uom = "]")
```
Public Members

```cpp
counter_type type_
The type of the described counter.
```

```cpp
std::uint32_t version_
The version of the described counter using the 0xMMmmSSSS scheme
```

```cpp
counter_status status_
The status of the counter object.
```

```cpp
std::string fullname_
The full name of this counter.
```

```cpp
std::string helptext_
The full descriptive text for this counter.
```

```cpp
std::string unit_of_measure_
The unit of measure for this counter.
```

Private Functions

```cpp
void serialize (serialization::output_archive & ar, const unsigned int)
```

```cpp
void serialize (serialization::input_archive & ar, const unsigned int)
```

Friends

```cpp
friend hpx::performance_counters::hpx::serialization::access
```

```cpp
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:
```

```cpp
/objectname{parentinstancename::parentindex/instancename#instanceindex} /counter-name#parameters
i.e. /queue{localityprefix/thread#2}/length
```

Public Types

```cpp
typedef counter_type_path_elements base_type
```

Public Functions

```cpp
counter_path_elements ()
```

```cpp
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)
```
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, const unsigned int)

void serialize (serialization::input_archive &ar, const unsigned int)

Friends

friend hpx::performance_counters::hpx::serialization::access
member holds a base counter name

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()`
`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, const unsigned int)`
void `serialize(serialization::input_archive &ar, const unsigned int)`

Friends

`friend hpx::performance_counters::hpx::serialization::access`

Defines

HPX_PERFORMANCE_COUNTER_V1
namespace hpx

namespace performance_counters

Enums

define counter_type
Values:

`counter_text`
`counter_text` shows a variable-length text string. It does not deliver calculated values.
Formula: None Average: None Type: Text

`counter_raw`
`counter_raw` shows the last observed value only. It does not deliver an average.
Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

`counter_monotonically_increasing`
counter_average_base

counter_average_base is used as the base data (denominator) in the computation of time or count averages for the counter_average_count and counter_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: \( \text{SUM} (N) / x \)
Type: Instantaneous

counter_average_count

counter_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: \( \frac{(N_1 - N_0)}{(D_1 - D_0)} \), 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 interval.
Average: \( \frac{(Nx - N0)}{(Dx - D0)} \) Type: Average

counter_aggregating

counter_aggregating applies a function to an embedded counter instance. The embedded counter is usually evaluated repeatedly after a fixed (but configurable) time interval.

Formula: \( F(Nx) \)

counter_average_timer

counter_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: \( \left(\frac{(N_1 - N_0)}{F}\right) / (D_1 - D_0) \), 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: \( \left(\frac{(Nx - N0)}{F}\right) / (Dx - D0) \) Type: Average

counter_elapsed_time

counter_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: \( \frac{(D_0 - N_0)}{F} \), 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: \( \frac{(Dx - N0)}{F} \) Type: Difference

counter_histogram

counter_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.

counter_raw_values

counter_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.

counter_text
counter_text shows a variable-length text string. It does not deliver calculated values.

Formula: None Average: None Type: Text

counter_raw
counter_raw shows the last observed value only. It does not deliver an average.

Formula: None. Shows raw data as collected. Average: None Type: Instantaneous

counter_monotonically_increasing

counter_average_base
counter_average_base is used as the base data (denominator) in the computation of time or count averages for the counter_average_count and counter_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

counter_average_count
counter_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

counter_aggregating
counter_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)

counter_average_timer
counter_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

counter_elapsed_time
counter_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
**counter_histogram**

`counter_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.

**counter_raw_values**

`counter_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.

**enum counter_status**

Values:

- **status_valid_data**
  
  No error occurred, data is valid.

- **status_new_data**
  
  Data is valid and different from last call.

- **status_invalid_data**
  
  Some error occurred, data is not value.

- **status_already_defined**
  
  The type or instance already has been defined.

- **status_counter_unknown**
  
  The counter instance is unknown.

- **status_counter_type_unknown**
  
  The counter type is unknown.

- **status_generic_error**
  
  A unknown error occurred.

- **status_valid_data**
  
  No error occurred, data is valid.

- **status_new_data**
  
  Data is valid and different from last call.

- **status_invalid_data**
  
  Some error occurred, data is not value.

- **status_already_defined**
  
  The type or instance already has been defined.

- **status_counter_unknown**
  
  The counter instance is unknown.

- **status_counter_type_unknown**
  
  The counter type is unknown.

- **status_generic_error**
  
  A unknown error occurred.

**enum discover_counters_mode**

Values:
discover_counters_minimal
discover_counters_full

Functions

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.

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_func const &discover_counters, mode = discover_counters_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_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_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_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_minimal,
   error_code &ec = throws)

bool expand_counter_info (counter_info const &, discover_counter_func const &,
   error_code &)

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.

lcos::future<naming::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.

lcos::future<naming::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 & help_text, 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 & help_text, std::uint32_t & version, error_code & ec = throws)
Retrieve the meta data specific for the given counter instance.

struct counter_value

**Public Functions**

counter_value (std::int64_t value = 0, std::int64_t scaling = 1, bool scale_inverse = false)

template<typename T>
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)

void serialize (serialization::input_archive & ar, const unsigned int)

**Friends**

friend hpx::performance_counters::hpx::serialization::access

struct counter_values_array

Chapter 2. What’s so special about HPX?
Public Functions

`counter_values_array (std::int64_t scaling = 1, bool scale_inverse = false)`

`counter_values_array (std::vector<std::int64_t> &&values, std::int64_t scaling = 1, bool scale_inverse = false)`

`counter_values_array (std::vector<std::int64_t> const &values, std::int64_t scaling = 1, bool scale_inverse = false)`

```cpp
template<typename T>
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

```cpp
void serialize (serialization::output_archive &ar, const unsigned int)
```

```cpp
void serialize (serialization::input_archive &ar, const unsigned int)
```

Friends

```cpp
friend hpx::performance_counters::hpx::serialization::access
namespace hpx
```

```cpp
namespace agas
```
Functions

```cpp
void locality_namespace_register_counter_types (error_code &ec = throws)
```

namespace hpx

namespace performance_counters

Functions

```cpp
void install_counter (naming::id_type const &id, counter_info const &info, error_code &ec = throws)
```

Install a new performance counter in a way, which will uninstall it automatically during shutdown.

namespace hpx

namespace performance_counters

Functions

```cpp
counter_status install_counter_type (std::string const &name, 
                                     hpx::util::function_nonser<std::int64_t> bool
                                    > const &counter_value, std::string const &helptext = "", std::string const &uom = "",
                                    counter_type type = counter_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.

Return If successful, this function returns `status_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`).

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.

counter_status install_counter_type (std::string const &name, 
  hpx::util::function_nonser< 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 pro-
  vided 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 ex-
  pected 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.

Return If successful, this function returns status_valid_data, otherwise it will either throw an excep-
  tion or return an error_code from the enum counter_status (also, see note related to parameter 
  ec).

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.

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 shut-
  down.

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.

Return If successful, this function returns status_valid_data, otherwise it will either throw an excep-
  tion or return an error_code from the enum counter_status (also, see note related to parameter 
  ec).

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.
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.

Return
If successful, this function returns status_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).

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.

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.

Return
If successful, this function returns status_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).

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.
• **helpertext**: [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.

```cpp
namespace hpx

namespace performance_counters

Functions

`std::vector<performance_counter> discover_counters (std::string const &name, error_code &ec = throws)`

struct performance_counter : public components::client_base<performance_counter, server::base_performance_counter>

Public Types

using base_type = components::client_base<performance_counter, server::base_performance_counter>

Public Functions

`performance_counter ()`
`performance_counter (std::string const &name)`
`performance_counter (std::string const &name, hpx::id_type const &locality)`
`performance_counter (id_type const &id)`
`performance_counter (future<id_type> &&id)`
`performance_counter (hpx::future<performance_counter> &&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 = false)`
`counter_value get_counter_value (launch::sync_policy, bool reset = false, error_code &ec = throws)`
`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 = false)

counter_values_array get_counter_values_array (launch::sync_policy, bool reset = false, 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 ()

bool start (launch::sync_policy, error_code &ec = throws)

future<bool> stop ()

bool stop (launch::sync_policy, error_code &ec = throws)

future<void> reset ()

void reset (launch::sync_policy, error_code &ec = throws)

future<void> reinit (bool reset = true)

void reinit (launch::sync_policy, bool reset = true, error_code &ec = throws)

future<std::string> get_name () const

std::string get_name (launch::sync_policy, error_code &ec = throws) const

template<typename T>
future<T> get_value (bool reset = false)

template<typename T>
T get_value (launch::sync_policy, bool reset = false, error_code &ec = throws)

template<typename T>
future<T> get_value () const

template<typename T>
T get_value (launch::sync_policy, error_code &ec = throws) const

Private Static Functions

template<typename T>
static T extract_value (future<counter_value> &&value)

namespace hpx

namespace performance_counters

struct performance_counter_base
Subclassed by hpx::performance_counters::server::base_performance_counter
Public Functions

```cpp
template <typename... Args>
virtual ~performance_counter_base() = default

// Destructor, needs to be virtual to allow for clean destruction of derived objects

virtual counter_info get_counter_info() const = 0;

virtual counter_value get_counter_value(bool reset = false) = 0;

virtual counter_values_array get_counter_values_array(bool reset = false) = 0;

virtual void reset_counter_value() = 0;

virtual void set_counter_value(const counter_value& value) = 0;

virtual bool start() = 0;

virtual bool stop() = 0;

virtual void reinit(bool reset) = 0;
```

namespace hpx

```cpp
namespace performance_counters

class performance_counter_set

Public Functions

performance_counter_set(bool print_counters_locally = false)
  Create an empty set of performance counters.

performance_counter_set(std::string const &names, bool reset = false)
  Create a set of performance counters from a name, possibly containing wild-card characters

performance_counter_set(std::vector<std::string> const &names, bool reset = false)

void add_counters(std::string const &names, bool reset = false, error_code &ec = throws)
  Add more performance counters to the set based on the given name, possibly containing wild-card characters

void add_counters(std::vector<std::string> const &names, bool reset = false, error_code &ec = throws)

std::vector<counter_info> get_counter_infos() const
  Retrieve the counter infos for all counters in this set.

std::vector<hpx::future<counter_value>> get_counter_values(bool reset = false) const
  Retrieve the values for all counters in this set supporting this operation

std::vector<counter_value> get_counter_values(launch::sync_policy, bool reset = false, error_code &ec = throws) const
```
std::vector<hpx::future<counter_values_array>> get_counter_values_array (bool reset = false) const

Retrieve the array-values for all counters in this set supporting this operation.

std::vector<counter_values_array> get_counter_values_array (launch::sync_policy, bool reset = false, error_code &ec = throws) const

std::vector<hpx::future<void>> reset ()

Reset all counters in this set.

void reset (launch::sync_policy, error_code &ec = throws)

std::vector<hpx::future<bool>> start ()

Start all counters in this set.

bool start (launch::sync_policy, error_code &ec = throws)

std::vector<hpx::future<bool>> stop ()

Stop all counters in this set.

bool stop (launch::sync_policy, error_code &ec = throws)

std::vector<hpx::future<void>> reinit (bool reset = true)

Re-initialize all counters in this set.

void reinit (launch::sync_policy, bool reset = true, error_code &ec = throws)

void release ()

Release all references to counters in the set.

std::size_t size () const

Return the number of counters in this set.

template<typename T>
hpx::future<std::vector<T>> get_values (bool reset = false) const

template<typename T>
std::vector<T> get_values (launch::sync_policy, bool reset = false, error_code &ec = throws) const

std::size_t get_invocation_count () const

Protected Functions

bool find_counter (counter_info const &info, bool reset, error_code &ec)
Protected Static Functions

template<typename T>
static std::vector<T> extract_values (std::vector<hpx::future<counter_value>> &values)

Private Types

typedef lcos::local::spinlock mutex_type

Private Members

mutex_type mtx_
std::vector<counter_info> infos_
std::vector<naming::id_type> ids_
std::vector<std::uint8_t> reset_
std::uint64_t invocation_count_
bool print_counters_locally_

namespace hpx

namespace agas

Functions

void primary_namespace_register_counter_types (error_code &ec = throws)

namespace hpx

namespace util

class query_counters

Public Functions

query_counters (std::vector<std::string> const &names, std::vector<std::string> const &reset_names, std::int64_t interval, std::string const &dest, std::string const &form, std::vector<std::string> const &shortnames, bool csv_header, bool print_counters_locally, bool counter_types)

~query_counters ()

void start ()

void stop_evaluating_counters (bool terminate = false)

bool evaluate (bool force = false)
void \textbf{terminate} ()

void \textbf{start\_counters} (error\_code \& ec = \textit{throws})

void \textbf{stop\_counters} (error\_code \& ec = \textit{throws})

void \textbf{reset\_counters} (error\_code \& ec = \textit{throws})

void \textbf{reinit\_counters} (bool reset = true, error\_code \& ec = \textit{throws})

bool \textbf{evaluate\_counters} (bool reset = false, char const *description = nullptr, bool force = false, error\_code \& ec = \textit{throws})

**Protected Functions**

void \textbf{find\_counters} ()

bool \textbf{print\_raw\_counters} (bool destination\_is\_cout, bool reset, bool no\_output, char const *description, std::vector<performance\_counters::counter\_info> const \& infos, error\_code \& ec)

bool \textbf{print\_array\_counters} (bool destination\_is\_cout, bool reset, bool no\_output, char const *description, std::vector<performance\_counters::counter\_info> const \& infos, error\_code \& ec)

template<typename Stream> void \textbf{print\_headers} (Stream \& output, std::vector<performance\_counters::counter\_info> const \& infos)

template<typename Stream, typename Future> void \textbf{print\_values} (Stream * output, std::vector<Future>&&, std::vector<std::size_t> &\& indices, std::vector<performance\_counters::counter\_info> const \& infos)

template<typename Stream> void \textbf{print\_value} (Stream *out, performance\_counters::counter\_info const \& infos, performance\_counters::counter\_value const \& value)

template<typename Stream> void \textbf{print\_value} (Stream *out, performance\_counters::counter\_info const \& infos, performance\_counters::counter\_values\_array const \& value)

template<typename Stream> void \textbf{print\_name\_csv} (Stream \& out, std::string const \& name)

template<typename Stream> void \textbf{print\_value\_csv} (Stream *out, performance\_counters::counter\_info const \& infos, performance\_counters::counter\_value const \& value)

template<typename Stream> void \textbf{print\_value\_csv} (Stream *out, performance\_counters::counter\_info const \& infos, performance\_counters::counter\_values\_array const \& value)

template<typename Stream> void \textbf{print\_name\_csv\_short} (Stream \& out, std::string const \& name)
Private Types

typedef lcos::local::mutex mutex_type

Private Functions

query_counters *this_()

Private Members

mutex_type mtx_
std::vector<std::string> names_
std::vector<std::string> reset_names_
performance_counters::performance_counter_set counters_
std::string destination_
std::string format_
std::vector<std::string> counter_shortnames_
bool csv_header_
bool print_counters_locally_
bool counter_types_
interval_timer timer_

namespace hpx

namespace performance_counters

class registry

Public Functions

registry()

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.

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::util::function_nonser<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::util::function_nonser<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::util::function_nonser<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::util::function_nonser<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(naming::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, naming::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

typedef std::map<std::string, counter_data> counter_type_map_type

Private Members

counter_type_map_type countertypes_

struct counter_data

Public Functions

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_

namespace hpx

namespace agas

Functions

void symbol_namespace_register_counter_types(error_code &ec = throws)

namespace hpx

namespace performance_counters

Functions

void register_threadmanager_counter_types(threads::threadmanager &tm)

namespace hpx

namespace performance_counters

namespace server

template<typename Operation>

class arithmetics_counter : public hpx::performance_counters::server::base_performance_counter, pub
Public Types

template<>
using type_holder = arithmetics_counter

template<>
using base_type_holder = base_performance_counter

Public Functions

arithmetics_counter()

arithmetics_counter(counter_info const &info, std::vector<std::string> const &base_counter_names)

hpx::performance_counters::counter_value get_counter_value (bool reset = false)

Overloads from the base_counter base class.

bool start ()

bool stop ()

void reset_counter_value ()

void finalize ()

Private Types

template<>
using base_type = components::component_base<arithmetics_counter<Operation>>

Private Members

performance_counter_set counters_

namespace hpx

namespace performance_counters

namespace server


template<typename Statistic>
class arithmetics_counter_extended : public hpx::performance_counters::server::base_performance_counter

Public Types

template<>
using type_holder = arithmetics_counter_extended

template<>
using base_type_holder = base_performance_counter

Public Functions

arithmetics_counter_extended()

arithmetics_counter_extended(counter_info const &info,
std::vector<std::string> const &base_counter_names)

hpx::performance_counters::counter_value get_counter_value(bool reset = false)

Overloads from the base_counter base class.

bool start()

bool stop()

void reset_counter_value()

void finalize()

Private Types

template<>
using base_type = components::component_base<arithmetics_counter_extended<Statistic>>

Private Members

performance_counter_set counters_

namespace hpx

namespace performance_counters

namespace server

class base_performance_counter : public hpx::performance_counters::performance_counter_base, public hpx::performance_counters::server::elapsed_time_counter,
Subclassed by hpx::performance_counters::server::arithmetics_counter< Operation >,
Statistic >, hpx::performance_counters::server::arithmetics_counter_extended<

hpx::performance_counters::server::raw_counter, hpx::performance_counters::server::raw_values_counter,

hpx::performance_counters::server::statistics_counter< Statistic >
Public Types

```cpp
using wrapping_type = components::component<base_performance_counter>
using base_type_holder = base_performance_counter
```

Public Functions

```cpp
base_performance_counter()
base_performance_counter (counter_info const &info)
constexpr void finalize ()
    finalize() will be called just before the instance gets destructed
counter_info get_counter_info_nonvirt () const
counter_value get_counter_value_nonvirt (bool reset)
counter_values_array get_counter_values_array_nonvirt (bool reset)
void set_counter_value_nonvirt (counter_value const &info)
void reset_counter_value_nonvirt ()
bool start_nonvirt ()
bool stop_nonvirt ()
void reinit_nonvirt (bool reset)
```

```cpp
HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, 
    get_counter_info_nonvirt, 
    get_counter_info_action)
```

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 `get_counter_info_action` retrieves a performance counters information.

```cpp
HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, 
    get_counter_value_nonvirt, 
    get_counter_value_action)
```

The `get_counter_value_action` queries the value of a performance counter.

```cpp
HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, 
    get_counter_values_array_nonvirt, 
    get_counter_values_array_action)
```

The `get_counter_values_array_action` queries the value of a performance counter.

```cpp
HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, 
    set_counter_value_nonvirt, 
    set_counter_value_action)
```

The `set_counter_value_action`.

```cpp
HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, 
    reset_counter_value_nonvirt, 
    reset_counter_value_action)
```

The `reset_counter_value_action`.  

2.8. API reference 1525
HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, start_action)

The start_action.

HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, stop_action)

The stop_action.

HPX_DEFINE_COMPONENT_ACTION (base_performance_counter, reinit_action)

The reinit_action.

Public Static Functions

static components::component_type get_component_type ()
static void set_component_type (components::component_type t)

Protected Functions

void reset_counter_value ()

the following functions are not implemented by default, they will just throw

void set_counter_value (counter_value const&)

counter_value get_counter_value (bool)

counter_values_array get_counter_values_array (bool)

bool start ()

bool stop ()

void reinit (bool)

counter_info get_counter_info () const

Protected Attributes

hpx::performance_counters::counter_info info_

util::atomic_count invocation_count_

namespace hpx

namespace agas
Functions

\[
\text{naming::\texttt{gid\_type component\_namespace\_statistics\_counter (std::string\& name) const}}
\]

\[
\text{HPX\_DEFINE\_PLAIN\_ACTION (component\_namespace\_statistics\_counter, compo-}
\text{nent\_namespace\_statistics\_counter\_action)}
\]

namespace hpx

namespace performance\_counters

namespace server

class elapsed\_time\_counter : public hpx::performance\_counters::server::base\_performance\_counter

Public Types

using type\_holder = elapsed\_time\_counter

using base\_type\_holder = base\_performance\_counter

Public Functions

elapsed\_time\_counter ()

elapsed\_time\_counter (counter\_info const &info)

hpx::performance\_counters::counter\_value get\_counter\_value (bool reset)

void reset\_counter\_value ()

the following functions are not implemented by default, they will just throw

bool start ()

bool stop ()

void finalize ()

Private Types

using base\_type = components::component\_base<elapsed\_time\_counter>

namespace hpx

namespace agas
Functions

`naming::gid_type locality_namespace_statistics_counter (std::string const &name)`

`HPX_DEFINE_PLAIN_ACTION (locality_namespace_statistics_counter, `locality_namespace_statistics_counter_action)`

namespace hpx

namespace agas

Functions

`naming::gid_type primary_namespace_statistics_counter (std::string const &name)`

`HPX_DEFINE_PLAIN_ACTION (primary_namespace_statistics_counter, primary_namespace_statistics_counter_action)`

namespace hpx

namespace performance_counters

namespace server

class raw_counter : public hpx::performance_counters::server::base_performance_counter, public components::component_base<raw_counter>

Public Types

using type_holder = raw_counter

using base_type_holder = base_performance_counter

Public Functions

`raw_counter ()`

`raw_counter (counter_info const &info, hpx::util::function_nonser<std::int64_t> bool>`

`hpx::performance_counters::counter_value get_counter_value (bool reset = false)`

void reset_counter_value ()

    the following functions are not implemented by default, they will just throw

void finalize ()
Private Types

```cpp
using base_type = components::component_base<raw_counter>
```

Private Members

```cpp
hpx::util::function_nonser<std::int64_t (bool) > f_
bool reset_
```

nenamespace hpx

```
namespace performance_counters

namespace server

class raw_values_counter: public hpx::performance_counters::server::base_performance_counter, public
```

Public Types

```cpp
using type_holder = raw_values_counter
using base_type_holder = base_performance_counter
```

Public Functions

```cpp
raw_values_counter()
raw_values_counter(counter_info const &info, hpx::util::function_nonser<std::vector<std::int64_t> (bool) > f
hpx::performance_counters::counter_values_array get_counter_values_array (bool reset = false)

void reset_counter_value ()
    the following functions are not implemented by default, they will just throw

void finalize ()
```

Private Types

```cpp
using base_type = components::component_base<raw_values_counter>
```
Private Members

```cpp
namespace hpx

namespace performance_counters

namespace server

template<typename Statistic>
class statistics_counter : public hpx::performance_counters::server::base_performance_counter, public components::component_base<
```

Public Types

```cpp
typedef statistics_counter type_holder
typedef base_performance_counter base_type_holder
```

Public Functions

```cpp
statistics_counter()
statistics_counter(counter_info const &info, std::string const &base_counter_name, std::size_t parameter1, std::size_t parameter2, bool reset_base_counter)
```

```cpp
hpx::performance_counters::counter_value get_counter_value (bool reset = false)
```

Overloads from the base_counter base class.

```cpp
bool start ()
bool stop ()
void reset_counter_value ()

the following functions are not implemented by default, they will just throw

void on_terminate ()
void finalize ()
```

Protected Functions

```cpp
bool evaluate_base_counter(counter_value &value)
bool evaluate ()
bool ensure_base_counter ()
```
Private Types

typedef components::component_base<statistics_counter<Statistic>> base_type

typedef lcos::local::spinlock mutex_type

Private Functions

statistics_counter *this_()

Private Members

mutex_type mtx_
hpx::util::interval_timer timer_
std::string base_counter_name_
naming::id_type base_counter_id_
std::unique_ptr<detail::counter_type_from_statistic_base> value_

counter_value prev_value_

bool has_prev_value_

std::size_t parameter1_

std::size_t parameter2_

bool reset_base_counter_

namespace hpx

namespace agas

Functions

naming::gid_type symbol_namespace_statistics_counter (std::string const &name)

HPX_DEFINE_PLAIN_ACTION (symbol_namespace_statistics_counter, sym-

bol_namespace_statistics_counter_action)

resiliency_distributed

The contents of this module can be included with the header hpx/modules/resiliency_distributed.hpp.
These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are
using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only
including the module header hpx/modules/resiliency_distributed.hpp, not the particular header in
which the functionality you would like to use is defined. See Public API for a list of names that are part of the public
HPX API.

namespace hpx
namespace resiliency

namespace experimental

Functions

template<typename Action, typename ...Ts>
hpz::future<typename hpz::util::detail::invoke_deferred_result<Action, hpz::naming::id_type, Ts...>::type> tag_invoke(
async_replay_t, const std::vector<hpz::naming::id_type>& ids, Action&& action, Ts&&... ts)

namespace hpx

namespace resiliency

namespace experimental

Functions

template<typename Vote, typename Pred, typename Action, typename ...Ts>

template<typename Vote, typename Action, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke(async_replicate_vote_validate_t, const std::vector<hpx::naming::id_type>& ids, Vote&& vote, Action&& action, Ts&&... ts)

template<typename Pred, typename Action, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke(async_replicate_validate_t, const std::vector<hpx::naming::id_type>& ids, Pred&& pred, Action&& action, Ts&&... ts)

template<typename Action, typename ...Ts>
hpx::future<typename hpx::util::detail::invoke_deferred_result<Action, hpx::naming::id_type, Ts...>::type> tag_invoke(async_replicate_t, const std::vector<hpx::naming::id_type>& ids, Action&& action, Ts&&... ts)
The contents of this module can be included with the header `hpx/modules/runtime_components.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header `hpx/modules/runtime_components.hpp`, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names 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 to create and 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_enum`. The default for this parameter is `hpx::components::factory_enabled`.

**Defines**

`HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY` *(...)*

This macro is used to create and register a minimal component registry with Hpx.Plugin.

`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_2` *(ComponentType, componentname)*

`HPX_REGISTER_MINIMAL_COMPONENT_REGISTRY_DYNAMIC_3` *(ComponentType, componentname, state)*

`namespace hpx`

```
namespace components

    template<typename Component, factory_state_enum 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

bool get_component_info (std::vector<std::string> &fillini, std::string const &filepath, 
  bool is_static = false)
Return the ini-information for all contained components.

  Return Returns true if the parameter fillini has been successfully initialized with the registry data of all implemented in this module.

  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.

void register_component_type ()
Return the unique identifier of the component type this factory is responsible for.

  Return Returns the unique identifier of the component type this factory instance is responsible for. This function throws on any error.

  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).

namespace hpx

Functions

components::server::runtime_support *get_runtime_support_ptr ()

namespace hpx

namespace components

Functions

void console_error_sink (naming::id_type const &dst, std::exception_ptr const &e)

void console_error_sink (std::exception_ptr const &e)

namespace hpx

namespace components
Functions

```cpp
void console_logging (logging_destination dest, std::size_t level, std::string const &msg)
void cleanup_logging ()
void activate_logging ()
```

```cpp
namespace hpx
```

```cpp
namespace components
```

Functions

```cpp
template<typename Component, typename ...Ts>
future<naming::id_type> create_async (naming::id_type const &gid, Ts&&... vs)
```

Asynchronously create a new instance of a component.

```cpp
template<typename Component, typename ...Ts>
future<std::vector<naming::id_type>> bulk_create_async (naming::id_type const &gid, std::size_t count, Ts&&... vs)
```

```cpp
template<typename Component, typename ...Ts>
naming::id_type create (naming::id_type const &gid, Ts&&... vs)
```

```cpp
template<typename Component, typename ...Ts>
std::vector<naming::id_type> bulk_create (naming::id_type const &gid, std::size_t count, Ts&&... vs)
```

```cpp
template<typename Component, typename ...Ts>
future<naming::id_type> create_colocated_async (naming::id_type const &gid, Ts&&... vs)
```

```cpp
template<typename Component, typename ...Ts>
static naming::id_type create_colocated (naming::id_type const &gid, Ts&&... vs)
```

```cpp
template<typename Component, typename ...Ts>
static future<std::vector<naming::id_type>> bulk_create_colocated_async (naming::id_type const &gid, std::size_t count, Ts&&... vs)
```

```cpp
template<typename Component, typename ...Ts>
std::vector<naming::id_type> bulk_create_colocated (naming::id_type const &id, std::size_t count, Ts&&... vs)
```

```cpp
namespace hpx
```

```cpp
namespace components
```
Variables

```cpp
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.

```cpp
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

```cpp
constexpr default_distribution_policy()
```
Default-construct a new instance of a `default_distribution_policy`. This policy will represent one locality (the local locality).

```cpp
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

```cpp
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

```cpp
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

```cpp
template<typename Component, typename ...Ts>
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
Return A future holding the global address which represents the newly created object
Parameters
- `vs`: [in] The arguments which will be forwarded to the constructor of the new object.

```cpp
template<typename Component, typename ...Ts>
hpx::future<std::vector<bulk_locality_result>> bulk_create(std::size_t count, Ts&&... vs)
```
Create multiple objects on the localities associated by this policy instance

Note This function is part of the placement policy implemented by this class
Return A future holding the list of global addresses which represent the newly created objects
Parameters
• count: [in] The number of objects to create
• vs: [in] The arguments which will be forwarded to the constructors of the new objects.

```cpp
template<typename Action, typename ...Ts>
async_result<Action>::type async (launch policy, Ts&&... vs) const

template<typename Action, typename Callback, typename ...Ts>
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

bool apply (Continuation &&c, threads::thread_priority priority, Ts&&... vs) const
  Note This function is part of the invocation policy implemented by this class

bool apply (threads::thread_priority priority, Ts&&... vs) const

bool apply_cb (Continuation &&c, threads::thread_priority priority, Callback &&cb, Ts&&... vs) const
  Note This function is part of the invocation policy implemented by this class

bool apply_cb (threads::thread_priority priority, Callback &&cb, Ts&&... vs) const

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::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

template<>
using type = hpx::future<typename traits::promise_local_result<typename hpx::traits::extract_action<Action>...>::type>

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_3 (ComponentType, componentname, basecomponentname)

HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC_4 (ComponentType, componentname, basecomponentname, state)

Defines

HPX_DISTRIBUTED_METADATA_DECLARATION (...)

HPX_DISTRIBUTED_METADATA_DECLARATION_1 (config)

HPX_DISTRIBUTED_METADATA_DECLARATION_2 (config, name)

HPX_DISTRIBUTED_METADATA (...)

HPX_DISTRIBUTED_METADATA_1 (config)

HPX_DISTRIBUTED_METADATA_2 (config, name)

namespace hpx

namespace components

namespace server

template<typename ConfigData, typename Derived = detail::this_type>
class distributed_metadata_base : public hpx::components::component_base<
std::conditional<
std::is_

Public Functions

distributed_metadata_base ()

distributed_metadata_base (ConfigData const &data)

ConfigData get () const
    Retrieve the configuration data.

HPX_DEFINE_COMPONENT_DIRECT_ACTION (distributed_metadata_base, get)
Private Members

ConfigData data_

namespace hpx

Functions

\[
\text{template<typename Component, typename... Ts><unspecified> hpx::new\_\_id\_type const & locality, Ts &&... vs)}
\]

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:

\[
\begin{align*}
\text{hpx::future<hpx::id\_type> f =} \\
\text{hpx::new\_<some\_component>(hpx::find\_here(), ...);} \\
\text{hpx::id\_type id = f.get();}
\end{align*}
\]

**Return** 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.

**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.

\[
\text{template<typename Component, typename... Ts><unspecified> hpx::local\_new(Ts &&... vs)}
\]

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:

\[
\begin{align*}
\text{hpx::future<hpx::id\_type> f =} \\
\text{hpx::local\_new\_<some\_component>(...);} \\
\text{hpx::id\_type id = f.get();}
\end{align*}
\]

**Return** 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 \texttt{hpx::future} object instance which can be used to retrieve the global address of the newly created component. If the first argument is \texttt{hpx::launch::sync} the function will directly return an \texttt{hpx::id\_type}.

- If the explicit template argument \texttt{Component} represents a client side object (\texttt{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.

\textbf{Note} The difference of this function to \texttt{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**

- \texttt{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.

\begin{verbatim}
template<typename Component, typename... Ts><unspecified> hpx::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.
\end{verbatim}

\textbf{Note} This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```cpp
hpx::future<std::vector<hpx::id_type> > f =
    hpx::new_<some_component[]>(hpx::find_here(), 10, ...);
locality id = f.get();
```

**Return** The function returns different types depending on its use:

- If the explicit template argument \texttt{Component} represents an array of a component type (i.e. \texttt{Component[]}, where \texttt{traits::is\_component<
Component>::value} evaluates to true), the function will return an \texttt{hpx::future} object instance which holds a \texttt{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 \texttt{Component} represents an array of a client side object type (i.e. \texttt{Component[]}, where \texttt{traits::is\_client<
Component>::value} evaluates to true), the function will return an \texttt{hpx::future} object instance which holds a \texttt{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.

**Parameters**

- \texttt{locality: [in]} The global address of the locality where the new instance should be created on.
- \texttt{count: [in]} The number of component instances to create
- \texttt{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.

\begin{verbatim}
template<typename Component, typename DistPolicy, typename... Ts><unspecified> hpx::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).
\end{verbatim}
Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```cpp
hpx::future<hpx::id_type> f =
    hpx::new_<some_component>(hpx::default_layout, ...);
hpx::id_type id = f.get();
```

Return 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.

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.

```cpp
template<typename Component, typename DistPolicy, typename... Ts><unordered> hpx::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:

```cpp
hpx::future<std::vector<hpx::id_type> > f =
    hpx::new_<some_component[]>(hpx::default_layout, 10, ...);
hpx::id_type id = f.get();
```

Return 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.

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.

```cpp
namespace hpx

namespace components

namespace server

Functions

void console_error_sink (std::exception_ptr const&)

HPX_DEFINE_PLAIN_ACTION (console_error_sink, console_error_sink_action)

namespace hpx

namespace components

namespace server

Functions

console_error_dispatcher & get_error_dispatcher ()

class console_error_dispatcher

Public Types

typedef util::spinlock mutex_type

typedef util::function_nonser<void (std::string const&)> sink_type

Public Functions

HPX_NON_COPYABLE (console_error_dispatcher)

console_error_dispatcher ()

template<typename F>
sink_type set_error_sink (F && sink)

void operator () (std::string const & msg)
```
Private Members

mutex_type mtx_
sink_type sink_

namespace hpx

namespace components

Typedefs

typedef hpx::tuple<logging_destination, std::size_t, std::string> message_type
typedef std::vector<message_type> messages_type

namespace server

Functions

void console_logging (messages_type const&)

template<typename Dummy = void>
class console_logging_action : public actions::direct_action<void (*) (messages_type const&),
  console_logging, console_logging_action<void>>

Public Functions

console_logging_action ()
console_logging_action (messages_type const &msgs)
console_logging_action (threads::thread_priority, messages_type const &msgs)

Public Static Functions

template<typename T>
static util::unused_type execute_function (naming::address_type, naming::component_type, T &v)
### Private Types

```cpp
typedef actions::direct_action<void (*) (messages_type const&), console_logging, console_logging_action> base_type
```

#### runtime_distributed

The contents of this module can be included with the header `hpx/modules/runtime_distributed.hpp`. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we **strongly** suggest only including the module header `hpx/modules/runtime_distributed.hpp`, not the particular header in which the functionality you would like to use is defined. See **Public API** for a list of names that are part of the public `HPX` API.

```cpp
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

```cpp
hpx::runtime_distributed::runtime_distributed(util::runtime_configuration & rtcfg, int(*pre_main)(runtime_mode) = 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 (util::function<nonser<hpx_main_function_type> const &func, bool blocking = false)

Start the runtime system.

Return 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.

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.

int start (bool blocking = false)

Start the runtime system.

Return 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.

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.

```cpp
int wait ()
    Wait for the shutdown action to be executed.
```

**Return** This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

```cpp
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.

```cpp
int finalize (double shutdown_timeout)
```

```cpp
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.

```cpp
int suspend ()
    Suspend the runtime system.
```

```cpp
int resume ()
    Resume the runtime system.
```

```cpp
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] Kill all localities attached to the currently running application (default: `true`)

```cpp
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] Kill all localities attached to the currently running application (default: true)

```cpp
int run (util::function_nonser<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.

**Return**  This function will return the value as returned as the result of the invocation of the function object given by the parameter `func`.

**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.

```cpp
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.

**Return**  This function will always return 0 (zero).

```cpp
bool is_networking_enabled ()
```

**template<typename F>**

```cpp
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.

```cpp
performance_counters::registry const &get_counter_registry () const
```

Allow access to the registry counter registry instance used by the HPX runtime.

```cpp
void register_counter_types ()
```

Install all performance counters related to this runtime instance.

```cpp
void register_query_counters (std::shared_ptr<util::query_counters> const &active_counters)
```

```cpp
void start_active_counters (error_code &ec = throws)
```

```cpp
void stop_active_counters (error_code &ec = throws)
```

```cpp
void reset_active_counters (error_code &ec = throws)
```

```cpp
void reinit_active_counters (bool reset = true, error_code &ec = throws)
```

```cpp
void evaluate_active_counters (bool reset = false, char const *description = nullptr, error_code &ec = throws)
```

```cpp
void stop_evaluating_counters (bool terminate = false)
```

```cpp
naming::resolver_client &get_agas_client ()
```

Allow access to the AGAS client instance used by the HPX runtime.
`hpx::threads::threadmanager &get_thread_manager()`  
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`  
Returns a string of the locality endpoints (usable in debug output)

`std::uint64_t 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)`  
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)`  
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)`  
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)`  
Add a function to be executed inside a HPX thread during `hpx::finalize`

**Parameters**
• \texttt{f}: The function \texttt{f} will be called from inside a HPX thread while \texttt{hpx::finalize} is executed. This is very useful to tear down the runtime environment of the application (uninstall performance counters, etc.)

\texttt{hpx::util::io_service_pool\*\texttt{get_thread_pool}(\texttt{char const}\ *\texttt{name})}

Access one of the internal thread pools (\texttt{io_service} instances) HPX is using to perform specific tasks. The three possible values for the argument \texttt{name} are “main_pool”, “io_pool”, “parcel_pool”, and “timer_pool”. For any other argument value the function will return zero.

\texttt{bool\ \texttt{register_thread}(\texttt{char const}\ *\texttt{name}, \texttt{std::size_t num} = 0, \texttt{bool service_thread} = \texttt{true}, error_code \&\texttt{ec} = \texttt{throws})}

Register an external OS-thread with HPX.

\texttt{notification_policy_type\ \texttt{get_notification_policy}(\texttt{char const}\ \*\texttt{prefix}, \texttt{runtime_local\::os_thread_type\ type})}

Generate a new notification policy instance for the given thread name prefix.

\texttt{std::uint32_t\ \texttt{get_locality_id}(error_code \&\texttt{ec}) const}

\texttt{std::size_t\ \texttt{get_num_worker_threads}() const}

\texttt{std::uint32_t\ \texttt{get_num_localities}(\texttt{hpx::launch\::sync_policy}, error_code \&\texttt{ec}) const}

\texttt{std::uint32_t\ \texttt{get_initial_num_localities}() const}

\texttt{lcos::future<\texttt{std::uint32_t}>\ \texttt{get_num_localities}() const}

\texttt{std::string\ \texttt{get_locality_name}() const}

\texttt{std::uint32_t\ \texttt{get_num_localities}(\texttt{hpx::launch\::sync_policy}, \texttt{components::component_type\ type}, error_code \&\texttt{ec}) const}

\texttt{lcos::future<\texttt{std::uint32_t}>\ \texttt{get_num_localities}(\texttt{components::component_type\ type})\ const}

\texttt{std::uint32_t\ \texttt{assign_cores}(\texttt{std::string const}\ \&\texttt{locality_basename}, \texttt{std::uint32_t num_threads})}

\texttt{std::uint32_t\ \texttt{assign_cores}()}

\textbf{Private Types}

\texttt{using\ used_cores_map_type = \texttt{std::map<\texttt{std::string, \texttt{std::uint32_t}>}}}

\textbf{Private Functions}

\texttt{threads::thread_result_type\ \texttt{run_helper}(\texttt{util::function_nonser<runtime::hpx_main_function_type>\ const}\ \&\texttt{func},\ \texttt{int}\ \&\texttt{result})}

\texttt{void\ \texttt{init_global_data}()}

\texttt{void\ \texttt{deinit_global_data}()}

\texttt{void\ \texttt{wait_helper}(\texttt{std::mutex \&mtx}, \texttt{std::condition_variable\ \&cond}, \texttt{bool\ \&running})}

\texttt{void\ \texttt{init_tss_helper}(\texttt{char const}\ \*\texttt{context}, \texttt{runtime_local\::os_thread_type\ type}, \texttt{std::size_t local_thread_num}, \texttt{std::size_t global_thread_num}, \texttt{char const}\ \*\texttt{pool_name}, \texttt{char const}\ \*\texttt{postfix}, \texttt{bool}\ \texttt{service_thread})}

\texttt{void\ \texttt{deinit_tss_helper}(\texttt{char const}\ \*\texttt{context}, \texttt{std::size_t num})}
void \texttt{init}\_tss\_ex}(std::string const &\texttt{locality}, char const *\texttt{context}, runtime\_local::os\_thread\_type \texttt{type}, std::size\_t \texttt{local\_thread\_num}, std::size\_t \texttt{global\_thread\_num}, char const *\texttt{pool\_name}, char const *\texttt{postfix}, bool service\_thread, error\_code &\texttt{ec})

\textbf{Private Members}

\begin{verbatim}
runtime_mode \texttt{mode}_
util::unique_id_ranges \texttt{id\_pool}_
naming::resolver\_client \texttt{agas\_client}_
applier::applier \texttt{applier}_
used\_cores\_map\_type \texttt{used\_cores\_map}_
std::unique\_ptr<components::server::runtime\_support> \texttt{runtime\_support}_
std::shared\_ptr<util::query\_counters> \texttt{active\_counters}_
int (*\texttt{pre\_main}_)(runtime\_mode)
\end{verbatim}

\textbf{Private Static Functions}

\begin{verbatim}
static void \texttt{default\_errorsink}(std::string const &)
namespace hpx
namespace applier
class applier
\end{verbatim}

\texttt{#include <applier.hpp>} The \texttt{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 \texttt{thread} will be created, if the target is remote a parcel will be sent.

\textbf{Public Functions}

\begin{verbatim}
HPX\_NON\_COPYABLE(applier)
applier()
void \texttt{init}(threads::threadmanager &\texttt{tm})
~applier()
void \texttt{initialize}(std::uint64\_t \texttt{rts})
threads::threadmanager &\texttt{get\_thread\_manager}()
Access the \texttt{thread\_manager} instance associated with this \texttt{applier}.
\end{verbatim}

This function returns a reference to the thread manager this applier instance has been created with.

\begin{verbatim}
naming::gid\_type \texttt{const &get\_raw\_locality}(error\_code &\texttt{ec} = \texttt{throws}) \texttt{const}
\end{verbatim}

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

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 = components::component_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.

Return The function returns true if there is at least one remote locality known to the AGAS service (!prefixes.empty()).

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.

bool get_remote_localities (std::vector<naming::id_type> &locality_ids, components::component_type type = components::component_invalid, error_code &ec = throws) const

bool get_raw_localities (std::vector<naming::gid_type> &locality_ids, components::component_type type = components::component_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.

Return The function returns true if there is at least one remote locality known to the AGAS service (!prefixes.empty()).

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.

bool get_localities (std::vector<naming::id_type> &locality_ids, error_code &ec = throws) const

bool get_localities (std::vector<naming::id_type> &locality_ids, components::component_type type, error_code &ec = throws) const

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

naming::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_

naming::id_type runtime_support_id_

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_ptr 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.

namespace hpx

namespace components

Functions

template<typename Component>
future<naming::id_type> copy(naming::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.

Return A future representing the global id of the newly (copied) 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.

template<typename Component>
future<naming::id_type> copy(naming::id_type const &to_copy, naming::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.
Return A future representing the global id of the newly (copied) 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.

template<typename Derived, typename Stub>
Derived copy (client_base<Derived, Stub> const &to_copy, naming::id_type const &target_locality = naming::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.

Return A future representing the global id of the newly (copied) 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.

namespace hpx

Functions

naming::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.

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).

Return The global id representing the root locality for this application.

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::naming::invalid_id otherwise.

See hpx::find_all_localities(), hpx::find_locality()

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::vector<naming::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.

**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).

**Return** The global ids representing the localities currently available to this application.

**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.

**See** `hpx::find_here()`, `hpx::find_locality()`

**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.

```cpp
class std::vector<naming::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).

**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).

**Return** The global ids representing the remote localities currently available to this application.

**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.

**See** `hpx::find_here()`, `hpx::find_locality()`

**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.

```cpp
namespace hpx
```

**Functions**

```cpp
class naming::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.

**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).
Return The global id representing the locality this function has been called 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 This function will return meaningful results only if called from an HPX-thread. It will return hpx::naming::invalid_id otherwise.

See hpx::find_all_localities(), hpx::find_locality()

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.

namespace hpx

Functions

std::vector<naming::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.

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).

Return 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.

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.

See hpx::find_here(), hpx::find_locality()

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.

std::vector<naming::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.
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).

Return The global ids representing the remote localities currently available to this application.

Note As long as \textit{ec} is not pre-initialized to \texttt{hpx::throws} this function doesn’t throw but returns the result code using the parameter \textit{ec}. Otherwise it throws an instance of \texttt{hpx::exception}.

Note This function will return meaningful results only if called from an HPX-thread. It will return an empty vector otherwise.

See \texttt{hpx::find_here()}, \texttt{hpx::find_locality()}

Parameters

- \texttt{type}: [in] The type of the components for which the function should return the available remote localities.
- \texttt{ec}: [in,out] this represents the error status on exit, if this is pre-initialized to \texttt{hpx::throws} the function will throw on error instead.

\texttt{naming::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.

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).

Return 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 \texttt{hpx::naming::invalid\_id}.

Note As long as \textit{ec} is not pre-initialized to \texttt{hpx::throws} this function doesn’t throw but returns the result code using the parameter \textit{ec}. Otherwise it throws an instance of \texttt{hpx::exception}.

Note This function will return meaningful results only if called from an HPX-thread. It will return \texttt{hpx::naming::invalid\_id} otherwise.

See \texttt{hpx::find\_here()}, \texttt{hpx::find\_all\_localities()}

Parameters

- \texttt{type}: [in] The type of the components for which the function should return any available locality.
- \texttt{ec}: [in,out] this represents the error status on exit, if this is pre-initialized to \texttt{hpx::throws} the function will throw on error instead.

\texttt{namespace hpx}
Functions

`future<std::string> get_locality_name(naming::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.

**Return** 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.

**See** `std::string get_locality_name()`

**Parameters**
- `id`: [in] The global id of the locality for which the name should be retrieved

namespace hpx

Functions

`lcos::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.

**Note** This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

**See** `hpx::find_all_localities, hpx::get_num_localities`

**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.

**Note** This function will return meaningful results only if called from an HPX-thread. It will return 0 otherwise.

**See** `hpx::find_all_localities, hpx::get_num_localities`

**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.
namespace components

Functions

template<typename Component, typename DistPolicy>
future<naming::id_type> migrate (naming::id_type 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.

Return A future representing the global id of the migrated component instance. This should be the same as migrate_to.

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.

template<typename Derived, typename Stub, typename DistPolicy>
Derived migrate (client_base<Derived, Stub> 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.

Return A future representing the global id of the migrated component instance. This should be the same as migrate_to.

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.

template<typename Component>
future<naming::id_type> migrate (naming::id_type const &to_migrate, naming::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.

Return A future representing the global id of the migrated component instance. This should be the same as migrate_to.

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.
template<typename Derived, typename Stub>

**Derived migrate** *(client_base<Derived, Stub> const &to_migrate, naming::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.

**Return** A client side representation of representing of the migrated component instance. This should be the same as `migrate_to`.

**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.

---

namespace hpx

namespace components

**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**

**runtime_support** *(naming::id_type const &gid = naming::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> naming::id_type create_component (Ts&&... vs)**

Create a new component type using the `runtime_support`.

**template<typename Component, typename ...Ts> lcos::future<naming::id_type> create_component_async (Ts&&... vs)**

Asynchronously create a new component using the `runtime_support`.

**template<typename Component, typename ...Ts> std::vector<naming::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> lcos::future<std::vector<naming::id_type>> bulk_create_components_async (std::size_t, Ts&&... vs)**

Asynchronously create a new component using the `runtime_support`.

**lcos::future<int> load_components_async ()**

**int load_components ()**

**lcos::future<void> call_startup_functions_async (bool pre_startup)**

**void call_startup_functions (bool pre_startup)**
lcos::future<void> shutdown_async (double timeout = -1)
    Shutdown the given runtime system.

void shutdown (double timeout = -1)
void shutdown_all (double timeout = -1)
    Shutdown the runtime systems of all localities.

lcos::future<void> terminate_async ()
    Terminate the given runtime system.

void terminate ()
void terminate_all ()
    Terminate the runtime systems of all localities.

void get_config (util::section & ini)
    Retrieve configuration information.

naming::id_type const &get_id () const
naming::gid_type const &get_raw_gid () const

Private Types

typedef stubs::runtime_support base_type

Private Members

naming::id_type gid_

namespace hpx

namespace components

namespace server

Functions

template<typename Component>
future<naming::id_type> copy_component_here (naming::id_type const &to_copy)

template<typename Component>
future<naming::id_type> copy_component (naming::id_type const &to_copy, naming::id_type const &target_locality)

namespace hpx

namespace components

namespace server
Functions

template<typename Component, typename DistPolicy>
future<id_type> migrate_component (id_type const &to_migrate, naming::address const &addr, DistPolicy const &policy)

template<typename Component, typename DistPolicy>
future<id_type> trigger_migrate_component (id_type const &to_migrate, DistPolicy const &policy, naming::id_type const &id, naming::address const &addr)

template<typename Component, typename DistPolicy>
future<id_type> perform_migrate_component (id_type const &to_migrate, DistPolicy const &policy)

namespace hpx

namespace components

namespace server

class runtime_support

Public Types

typedef runtime_support type_holder

Public Functions

runtime_support (hpx::util::runtime_configuration &cfg)

~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 local_op)`

`template<typename Component> naming::gid_type migrate_component_to_here (std::shared_ptr<Component> const & p, naming::id_type to_migrate)`

`void shutdown (double timeout, naming::id_type const & respond_to)`
Gracefully shutdown this runtime system instance.

`void shutdown_all (double timeout)`
Gracefully shutdown runtime system instances on all localities.

`HPX_NORETURN void hpx::components::server::runtime_support::terminate(naming::id_type const & respond_to)`
Shutdown this runtime system instance.

`void terminate_act (naming::id_type const & id)`

`HPX_NORETURN void hpx::components::server::runtime_support::terminate_all()`
Shutdown runtime system instances on all localities.

`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, load_components)`
termination detection

`HPX_DEFINE_COMPONENT_ACTION (runtime_support, call_startup_functions)`

`HPX_DEFINE_COMPONENT_ACTION (runtime_support, call_shutdown_functions)`

`HPX_DEFINE_COMPONENT_ACTION (runtime_support, shutdown)`

`HPX_DEFINE_COMPONENT_ACTION (runtime_support, shutdown_all)`

`HPX_DEFINE_COMPONENT_ACTION (runtime_support, terminate_act, terminate_action)`

`HPX_DEFINE_COMPONENT_ACTION (runtime_support, terminate_all_act, terminate_all_action)`
HPX_DEFINE_COMPONENT_DIRECT_ACTION (runtime_support, get_config)

HPX_DEFINE_COMPONENT_ACTION (runtime_support, garbage_collect)

HPX_DEFINE_COMPONENT_ACTION (runtime_support, create_performance_counter)

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, naming::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 ()

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 Static Functions

static component_type get_component_type ()

static void set_component_type (component_type t)

static 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 bool is_target_valid (naming::id_type const &id)
Chapter 2. What’s so special about HPX?

Protected Functions

```cpp
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 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 (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<naming::id_type> const &locality_ids)
```
Private Types

typedef lcos::local::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_
lcos::local::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

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

namespace hpx

namespace components

namespace stubs

struct runtime_support
  Subclassed by hpx::components::runtime_support

Public Static Functions

template<typename Component, typename ...Ts>
static lcos::future<naming::id_type> create_component_async (naming::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 naming::id_type create_component (naming::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 lcos::future<std::vector<naming::id_type>> bulk_create_component_colocated_async (naming::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 std::vector<naming::id_type> bulk_create_component_colocated (naming::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 lcos::future<std::vector<naming::id_type>> bulk_create_component_async (naming::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 std::vector<naming::id_type> bulk_create_component (naming::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 lcos::future<naming::id_type> create_component_colocated_async (naming::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 naming::id_type create_component_colocated (naming::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 lcos::future<naming::id_type> copy_create_component_async (naming::id_type const &gid,
std::shared_ptr<Component> const &p, bool local_op)

template<typename Component>
static naming::id_type copy_create_component (naming::id_type const &gid,
std::shared_ptr<Component> const &p, bool local_op)
static lcos::future<naming::id_type> migrate_component_async(naming::id_type const &target_locality, std::shared_ptr<Component> const &p, naming::id_type const &to_migrate)

template<typename Component, typename DistPolicy>
static lcos::future<naming::id_type> migrate_component_async(DistPolicy const &policy, std::shared_ptr<Component> const &p, naming::id_type const &to_migrate)

template<typename Component, typename Target>
static naming::id_type migrate_component(Target const &target, naming::id_type const &to_migrate, std::shared_ptr<Component> const &p)

static lcos::future<int> load_components_async(naming::id_type const &gid)
static int load_components(naming::id_type const &gid)

static lcos::future<void> call_startup_functions_async(naming::id_type const &gid, bool pre_startup)
static void call_startup_functions(naming::id_type const &gid, bool pre_startup)

static lcos::future<void> shutdown_async(naming::id_type const &targetgid, double timeout = -1)
Shut down the given runtime system.

static void shutdown(naming::id_type const &targetgid, double timeout = -1)
static void shutdown_all(naming::id_type const &targetgid, double timeout = -1)
Shut down the runtime systems of all localities.

static void shutdown_all(double timeout = -1)

static lcos::future<void> terminate_async(naming::id_type const &targetgid)
Retrieve configuration information.

static void terminate(naming::id_type const &targetgid)
static void terminate_all(naming::id_type const &targetgid)

static void terminate_all()
static lcos::future<void> garbage_collect_async (naming::id_type const &targetgid)

static void garbage_collect (naming::id_type const &targetgid)

static lcos::future<naming::id_type> create_performance_counter_async (naming::id_type targetgid, performance_counters::counter_info const &info)

static naming::id_type create_performance_counter (naming::id_type targetgid, performance_counters::counter_info const &info, error_code &ec = throws)

static lcos::future<util::section> get_config_async (naming::id_type const &targetgid)

Retrieve configuration information.

static void get_config (naming::id_type const &targetgid, util::section &ini)

static void remove_from_connection_cache_async (naming::id_type const &target, naming::gid_type const &gid, parcelset::endpoints_type const &endpoints)

segmented_algorithms

The contents of this module can be included with the header hpx/modules/segmented_algorithms.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/segmented_algorithms.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace hpx

    namespace segmented
Functions

template<typename ExPolicy, typename FwdIter1, typename FwdIter2, typename Op>
parallel::util::detail::algorithm_result<ExPolicy, FwdIter2>::type 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)

namespace hpx

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)

namespace hpx

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>
namespace hpx

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_invoke(
    hpx::count_if_t,
    ExPolicy &&policy,
    SegIter first,
    SegIter last,
    T &&f
)

2.8. API reference
namespace hpx

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())

namespace hpx

namespace segmented

Functions

template<typename SegIter, typename T>
SegIter tag_invoke(hpx::find_t, SegIter first, SegIter last, T const &val)

template<typename ExPolicy, typename SegIter, typename T>
parallel::util::detail::algorithm_result<ExPolicy, SegIter>::type tag_invoke(hpx::exclusive_scan_t, ExPolicy &&policy, SegIter first, SegIter last, T init, Op &&op = Op())

template<typename FwdIter, typename F>
FwdIter tag_invoke(hpx::find_if_t, FwdIter first, FwdIter last, F &&f)

template<typename ExPolicy, typename FwdIter, typename F>

namespace hpx

namespace segmented

Functions

template<
typename InIter, typename F>
InIter tag_invoke(hpx::for_each_t, InIter first, InIter last, 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 &


template<
typename InIter, typename Size, typename F>
InIter tag_invoke(hpx::for_each_n_t, InIter first, Size count, 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 &

namespace hpx

namespace segmented
Functions

template<typename SegIter, typename F>
SegIter tag_invoke(hpx::generate_t, SegIter first, SegIter last, F &f)

namespace hpx

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())

namespace hpx

namespace segmented

Functions

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())

namespace hpx

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)

namespace hpx

namespace segmented

Functions
namespace hpx

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_invoke(hpx::transform_t, ExPolicy &&policy, SegIter first, SegIter last, OutIter dest, F &&f)

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 tag_invoke(hpx::transform_t, ExPolicy &&policy, InIter1 first1, InIter1 last1, InIter2 first2, OutIter dest, F &&f)
namespace hpx

namespace segmented

Functions

template<
    typename InIter1,
    typename InIter2,
    typename OutIter,
    typename F
>
hp::parallel::util::in_in_out_result<InIter1, InIter2, OutIter>
tag_invoke(hp::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
>
hp::parallel::util::detail::algorithm_result<ExPolicy, hp::parallel::util::in_in_out_result<InIter1, InIter2, OutIter>>::type
tag_invoke(hp::transform_t,
    ExPolicy &&policy, InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, OutIter dest, F &&f)

namespace hpx

namespace segmented

1576 Chapter 2. What’s so special about HPX?
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)

namespace hpx

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>
namespace hpx

namespace util

namespace functional

struct segmented_iterator_begin

    template<typename Iterator>
    struct apply
Public Types

```cpp
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_iterator
```

Public Functions

```cpp
template<typename SegIter>
type operator() (SegIter iter) const
```

```cpp
struct segmented_iterator_end
```

```cpp
template<typename Iterator>
struct apply
```

Public Types

```cpp
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_iterator
```

Public Functions

```cpp
template<typename SegIter>
type operator() (SegIter iter) const
```

```cpp
struct segmented_iterator_local
```

```cpp
template<typename Iterator>
struct apply
```

Public Types

```cpp
template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_iterator
```

Public Functions

```cpp
template<typename Iter>
type operator() (Iter iter) const
```

```cpp
struct segmented_iterator_local_begin
```

```cpp
template<typename Iterator>
struct apply
```
Public Types

template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_raw_iterator

Public Functions

template<typename LocalSegIter>
type operator() (LocalSegIter iter) const

struct segmented_iterator_local_end

Public Types

template<>
using type = typename traits::segmented_iterator_traits<Iterator>::local_raw_iterator

Public Functions

template<typename LocalSegIter>
type operator() (LocalSegIter iter) const

struct segmented_iterator_segment

Public Types

template<>
using type = typename traits::segmented_iterator_traits<Iterator>::segment_iterator

Public Functions

template<typename Iter>
type operator() (Iter iter) const

template<typename ...Ts>
struct segmented_iterator_traits<util::zip_iterator<Ts...>, typename std::enable_if<util::all_of<typename segmented_iterator_traits<Ts>::is_segmented_iterator...>::value>::type>
Public Types

```cpp
typedef std::true_type is_segmented_iterator
typedef util::zip_iterator<Ts...> iterator

typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::segment_iterator...> segment_iterator

typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_segment_iterator...> local_segment_iterator

typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_iterator...> local_iterator

typedef util::zip_iterator<typename segmented_iterator_traits<Ts>::local_raw_iterator...> local_raw_iterator
```

Public Static Functions

```cpp
static segment_iterator segment (iterator iter)
static local_iterator local (iterator iter)
static local_iterator begin (segment_iterator const &iter)
static local_iterator end (segment_iterator const &iter)
static local_raw_iterator begin (local_segment_iterator const &seg_iter)
static local_raw_iterator end (local_segment_iterator const &seg_iter)
static naming::id_type get_id (segment_iterator const &iter)
```

```cpp
template<typename ...Ts>
struct segmented_local_iterator_traits<util::zip_iterator<Ts...>, typename std::enable_if<util::all_of<typename segmented_local_iterator_traits<Ts>::is_segmented_local_iterator...>::value>::type>
```

```
2.8. API reference 1581
```
Public Types

typedef std::true_type is_segmented_iterator
typedef util::zip_iterator<Ts...> iterator
typedef util::zip_iterator<
typename segmented_iterator_traits<Ts>::segment_iterator...> segment_iterator
typedef util::zip_iterator<
typename segmented_iterator_traits<Ts>::local_segment_iterator...> local_segment_iterator
typedef util::zip_iterator<
typename segmented_iterator_traits<Ts>::local_iterator...> local_iterator
typedef util::zip_iterator<
typename segmented_iterator_traits<Ts>::local_raw_iterator...> local_raw_iterator

Public Static Functions

static segment_iterator segment (iterator iter)
static local_iterator local (iterator iter)
static local_iterator begin (segment_iterator const &iter)
static local_iterator end (segment_iterator const &iter)
static local_raw_iterator begin (local_segment_iterator const &seg_iter)
static local_raw_iterator end (local_segment_iterator const &seg_iter)
static naming::id_type get_id (segment_iterator const &iter)

template<typename ...Ts>
struct segmented_local_iterator_traits<util::zip_iterator<Ts...>, typename std::enable_if<util::all_of<typename segmented_iterator_traits<Ts>::is_segmented_iterator...>::value>::type>

Public Types

typedef std::true_type is_segmented_local_iterator
typedef util::zip_iterator<Ts...> iterator
typedef util::zip_iterator<Ts...> local_iterator
typedef util::zip_iterator<Ts...> local_raw_iterator

Public Static Functions

static local_raw_iterator local (local_iterator const &iter)
static local_iterator remote (local_raw_iterator const &iter)

namespace functional

struct get_raw_iterator

template<typename Iterator>
struct apply
Public Functions

template<typename SegIter>
segmented_iterator_traits<Iter>::local_raw_iterator operator() (SegIter iter) const

struct get_remote_iterator

template<typename Iterator>
struct apply

Public Functions

template<typename SegIter>
segmented_iterator_traits<Iter>::local_iterator operator() (SegIter iter) const

statistics

The contents of this module can be included with the header hpx/modules/statistics.hpp. These headers may be used by user-code but are not guaranteed stable (neither header location nor contents). You are using these at your own risk. If you wish to use non-public functionality from a module we strongly suggest only including the module header hpx/modules/statistics.hpp, not the particular header in which the functionality you would like to use is defined. See Public API for a list of names that are part of the public HPX API.

namespace boost

namespace accumulators

namespace extract

Variables

const extractor<tag::histogram> histogram = {}

namespace tag

struct histogram : public depends_on<count>, public histogram_num_bins, public histogram_min_range

struct impl

template<typename Sample, typename Weight>
struct apply
Public Types

typedef hpx::util::detail::histogram_impl<Sample> type

namespace boost

namespace accumulators

namespace extract

Variables

const extractor<tag::rolling_max> rolling_max = {}

namespace tag

struct rolling_max : public depends_on<rolling_window>

struct impl

template<
typename Sample, typename Weight>

struct apply

Public Types

typedef hpx::util::detail::rolling_max_impl<Sample> type

namespace boost

namespace accumulators

namespace extract

Variables

const extractor<tag::rolling_min> rolling_min = {}

namespace tag

struct rolling_min : public depends_on<rolling_window>

struct impl

template<
typename Sample, typename Weight>

struct apply
Public Types

typedef hpx::util::detail::rolling_min_impl<Sample> type

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\textsuperscript{212}. 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\textsuperscript{213} 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.

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

\textsuperscript{212} https://github.com/STEllAR-GROUP/hpx/blob/master/.github/CONTRIBUTING.md
\textsuperscript{213} http://hpx.stellar-group.org/documents/governance/
5. This step does not apply to patch releases. For both APEX and libCDS:
   - Change the release branch to be the most current release tag available in the APEX/libCDS
git_external section in the main CMakeLists.txt. Please contact the maintainers of the respec-
tive packages to generate a new release to synchronize with the HPX release (APEX\textsuperscript{214}, libCDS\textsuperscript{215}).

6. Make sure \texttt{HPX_VERSION\_MAJOR/MINOR/SUBMINOR} in CMakeLists.txt contain the correct values. Change them if needed.

7. 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).

8. Update the minimum required versions if necessary (compilers, dependencies, etc.) in building_hpx.rst.

9. Verify that the Jenkins setups for the release branch on Rostam and Piz Daint are running and do not display any
   errors.

10. Repeat the following steps until satisfied with the release.

   1. Change \texttt{HPX\_VERSION\_TAG} in CMakeLists.txt to \texttt{-rcN}, where \texttt{N} is the current iteration of this
      step. Start with \texttt{-rc1}.

   2. Create a pre-release on GitHub using the script \texttt{tools/roll\_release.sh}. This script automatically
      tag with the corresponding release number. The script requires that you have the STEllAR Group signing
      key.

   3. This step is not necessary for patch releases. Notify \texttt{hpx-users@stellar\_group.org} and
      \texttt{stellar@cct.lsu.edu} 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.
      a. Use \texttt{git merge} when possible, and fall back to \texttt{git cherry\_pick} when needed. For patch
         releases \texttt{git cherry\_pick} is most likely your only choice if there have been significant unrelated
         changes on master since the previous release.
      b. Go back to the first step when enough patches have been added.
      c. If there are no more patches, continue to make the final release.

11. Update any occurrences of the latest stable release to refer to the version about to be released. For example,
    \texttt{quickstart.rst} contains instructions to check out the latest stable tag. Make sure that refers to the new
    version.

12. Add a new entry to the RPM changelog (\texttt{cmake/packaging/rpm/Changelog.txt}) with the new version
    number and a link to the corresponding changelog.

\textsuperscript{214} http://github.com/UO-OACISS/xpress-apex
\textsuperscript{215} https://github.com/STEllAR-GROUP/libcds
13. Change `HPX_VERSION_TAG` in `CMakeLists.txt` to an empty string.

14. 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`.

15. 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 STEllAR Group signing key.

16. Update the websites (hpx.stellar-group.org216, stellar-group.org217 and stellar.cct.lsu.edu218). 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).

17. Merge release branch into master.

18. 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`.

19. 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

   - Update version number in `hpx/package.py` and SHA256 to that of the new release

21. Announce the release on hpx-users@stellar-group.org, stellar@ct.c.lsu.edu, alllct@ct.c.lsu.edu, faculty@cse.lsu.edu, faculty@ece.lsu.edu, xpress@crest.iu.edu, the HPX Slack channel, the IRC channel, Sonia Sachs, our list of external collaborators, isocpp.org, reddit.com, HPC Wire, Inside HPC, Heise Online, and a CCT press release.

22. Beer and pizza.

### 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 CTest219 tool and are executed automatically on each commit to the HPX Github220 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 Issues221 tracker with detailed information about the target system.

---

216 https://hpx.stellar-group.org  
217 https://stellar-group.org  
218 https://stellar.cct.lsu.edu  
220 https://github.com/STEllAR-GROUP/hpx/  
221 https://github.com/STEllAR-GROUP/hpx/issues  

2.9. Contributing to HPX
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\(^{222}\) 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` for the executable name and `hpx_test_options` for the corresponding options to use for the run.

Issue tracker

If you stumble over a bug or missing feature in HPX, please submit an issue to our HPX Issues\(^ {223}\) 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\(^ {224}\) page.

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\(^ {225}\) and a combination of CDash\(^ {226}\) and pycicle\(^ {227}\). You can see the dashboards here: CircleCI HPX dashboard\(^ {228}\) and CDash HPX dashboard\(^ {229}\).

\(^{223}\) https://github.com/STEllAR-GROUP/hpx/issues
\(^{224}\) https://stellar.cct.lsu.edu/support/
\(^{225}\) https://circleci.com
\(^{226}\) https://www.kitware.com/cdash/project/about.html
\(^{227}\) https://github.com/biddisco/pycicle/
\(^{228}\) https://circleci.com/gh/STEllAR-GROUP/hpx
\(^{229}\) https://cdash.cscs.ch/index.php?project=HPX
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 provides a convenient alternative to quickly get all the dependencies needed to start development of HPX. Our testing setup on CircleCI uses a docker image to run all tests.

To get started you need to install Docker using whatever means is most convenient on your system. Once you have Docker 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. To start a container using the HPX build environment, run:

```bash
$ 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 for more information on using Docker.

**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)).

2.9.6 Documentation

This documentation is built using Sphinx, and an automatically generated API reference using Doxygen and Breathe.

We always welcome suggestions on how to improve our documentation, as well as pull requests with corrections and additions.

### Prerequisites

To build the HPX documentation, you need recent versions of the following packages:

- python3
- sphinx 3.5.4 (Python package)
- sphinx_rtd_theme (Python package)
- breathe 4.16.0 (Python package)
- doxygen

If the Python dependencies are not available through your system package manager, you can install them using the Python package manager pip:

```
230 https://www.docker.com
231 https://circleci.com
232 https://www.docker.com
233 https://www.docker.com
234 https://hub.docker.com/r/stellargroup/build_env/
235 https://docs.docker.com/
236 https://www.docker.com
237 https://docs.docker.com/storage/bind-mounts/
238 http://www.sphinx-doc.org
239 https://www.doxygen.org
240 https://breathe.readthedocs.io/en/latest
241 https://www.python.org
```
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\(^{242}\) tool.

**SPHINX_ROOT**:PATH
Specifies where to look for the installation of the Sphinx\(^{243}\) tool.

**BREATHE_API_DOC_ROOT**:PATH
Specifies where to look for the installation of the Breathe\(^{244}\) tool.

## Building documentation

Enable building of the documentation by setting `HPX_WITH_DOCUMENTATION=ON` during CMake\(^{245}\) 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.

### 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:`.
- 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.

\(^{242}\) [https://www.doxygen.org](https://www.doxygen.org)

\(^{243}\) [http://www.sphinx-doc.org](http://www.sphinx-doc.org)

\(^{244}\) [https://breathe.readthedocs.io/en/latest](https://breathe.readthedocs.io/en/latest)

\(^{245}\) [https://www.cmake.org](https://www.cmake.org)
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`\(^\text{246}\) 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/
      * `<lib_name>`
  - src/
    * CMakeLists.txt
  - tests/
    * CMakeLists.txt
    * unit/
      * CMakeLists.txt
    * regressions/
      * CMakeLists.txt
    * performance/
      * CMakeLists.txt

\(^{246}\) https://github.com/STEllAR-GROUP/hpx/blob/master/libs/create_library_skeleton.py
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 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. If a header should be excluded from the API reference, a comment `// sphinx:undocumented` needs to be added.

Tests are placed in suitable subdirectories of `tests`.

When in doubt, consult existing modules for examples on how to structure the module.

### 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` 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:

---

247 https://www.cmake.org

248 https://github.com/tomtom-international/cpp-dependencies

---
$ 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 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.
Closed issues

- Issue #5494\textsuperscript{249} - Rename \texttt{simdpar} execution policy to \texttt{par\_simd}
- Issue #5488\textsuperscript{250} - \texttt{hpx::util::bind} doesn’t bounds-check placeholders
- Issue #5486\textsuperscript{251} - Possible V1.7.1 release

Closed pull requests

- PR #5500\textsuperscript{252} - Minor bug fix in transform exclusive and inclusive scan tests
- PR #5499\textsuperscript{253} - Rename \texttt{simdpar} to \texttt{par\_simd}
- PR #5489\textsuperscript{254} - Adding bound-checking for \texttt{bind} placeholders
- PR #5485\textsuperscript{255} - Add Asio version check
- PR #5482\textsuperscript{256} - Change extra archive data to rely on uniqueness of a variable address, not a function address
- PR #5448\textsuperscript{257} - More fixes to enable for all types to be assumed to be bitwise copyable
- PR #5445\textsuperscript{258} - Improve performance of Spinlocks
- PR #5444\textsuperscript{259} - Adapt transform\_inclusive\_scan to C++ 20
- PR #5440\textsuperscript{260} - Adapt transform\_exclusive\_scan to C++ 20
- PR #5439\textsuperscript{261} - Adapt inclusive\_scan to C++ 20
- PR #5436\textsuperscript{262} - Adapt exclusive\_scan to C++20

2.10.2 \textit{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.

\textsuperscript{249} https://github.com/STEllAR-GROUP/hpx/issues/5494
\textsuperscript{250} https://github.com/STEllAR-GROUP/hpx/issues/5488
\textsuperscript{251} https://github.com/STEllAR-GROUP/hpx/issues/5486
\textsuperscript{252} https://github.com/STEllAR-GROUP/hpx/pull/5500
\textsuperscript{253} https://github.com/STEllAR-GROUP/hpx/pull/5499
\textsuperscript{254} https://github.com/STEllAR-GROUP/hpx/pull/5489
\textsuperscript{255} https://github.com/STEllAR-GROUP/hpx/pull/5485
\textsuperscript{256} https://github.com/STEllAR-GROUP/hpx/pull/5482
\textsuperscript{257} https://github.com/STEllAR-GROUP/hpx/pull/5448
\textsuperscript{258} https://github.com/STEllAR-GROUP/hpx/pull/5445
\textsuperscript{259} https://github.com/STEllAR-GROUP/hpx/pull/5444
\textsuperscript{260} https://github.com/STEllAR-GROUP/hpx/pull/5440
\textsuperscript{261} https://github.com/STEllAR-GROUP/hpx/pull/5439
\textsuperscript{262} https://github.com/STEllAR-GROUP/hpx/pull/5436

1594 Chapter 2. What’s so special about \textit{HPX}?
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,
  - 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,
  - 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/christophh/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\(^{263}\) - Fix lvalue-ref qualified connect for `when_all-sender`
- Issue #5412\(^{264}\) - Link error
- Issue #5397\(^{265}\) - Performance regression in thread annotations
- Issue #5395\(^{266}\) - HPX 1.7.0-rc1 fails to build icw APEX + OTF2
- Issue #5385\(^{267}\) - HPX 1.7 crashes on Piz Daint > 64 nodes
- Issue #5380\(^{268}\) - CMake should search for asio package installed on the system
- Issue #5378\(^{269}\) - HPX 1.7.0 stopped building on Fedora
- Issue #5369\(^{270}\) - HPX 1.6 and master hangs on Summit for > 64 nodes
- Issue #5358\(^{271}\) - HPX init fails for single-core environments
- Issue #5345\(^{272}\) - Rename P2220 property CPOs?
- Issue #5333\(^{273}\) - HPX does not compile on the new Mac OSX using the M1 chip
- Issue #5317\(^{274}\) - Consider making hipblas optional
- Issue #5306\(^{275}\) - asio fails to build with CUDA 10.0
- Issue #5294\(^{276}\) - `execution::on` should be based on `execution::schedule`
- Issue #5275\(^{277}\) - HPX V1.6.0 fails on Fedora release
- Issue #5270\(^{278}\) - HPX-1.6.0 fails to build on Windows 10
- Issue #5257\(^{279}\) - Allow triggering the output of OS thread affinity from configuration settings
- Issue #5246\(^{280}\) - HPX fails to build on ppc64le
- Issue #5232\(^{281}\) - Annotation using `hpx::util::annotated_function` not working
- Issue #5222\(^{282}\) - Build and link errors with ittnotify enabled
- Issue #5204\(^{283}\) - Move algorithms to `tag_fallbackdispatch`
- Issue #5163\(^{284}\) - Remove module-specific compatibility and deprecation options

[^263]: https://github.com/STEllAR-GROUP/hpx/issues/5423
[^264]: https://github.com/STEllAR-GROUP/hpx/issues/5412
[^265]: https://github.com/STEllAR-GROUP/hpx/issues/5397
[^266]: https://github.com/STEllAR-GROUP/hpx/issues/5395
[^267]: https://github.com/STEllAR-GROUP/hpx/issues/5385
[^268]: https://github.com/STEllAR-GROUP/hpx/issues/5380
[^269]: https://github.com/STEllAR-GROUP/hpx/issues/5369
[^270]: https://github.com/STEllAR-GROUP/hpx/issues/5358
[^271]: https://github.com/STEllAR-GROUP/hpx/issues/5345
[^272]: https://github.com/STEllAR-GROUP/hpx/issues/5333
[^273]: https://github.com/STEllAR-GROUP/hpx/issues/5317
[^274]: https://github.com/STEllAR-GROUP/hpx/issues/5306
[^275]: https://github.com/STEllAR-GROUP/hpx/issues/5294
[^276]: https://github.com/STEllAR-GROUP/hpx/issues/5275
[^277]: https://github.com/STEllAR-GROUP/hpx/issues/5270
[^278]: https://github.com/STEllAR-GROUP/hpx/issues/5257
[^279]: https://github.com/STEllAR-GROUP/hpx/issues/5246
[^280]: https://github.com/STEllAR-GROUP/hpx/issues/5232
[^281]: https://github.com/STEllAR-GROUP/hpx/issues/5222
[^282]: https://github.com/STEllAR-GROUP/hpx/issues/5204
[^283]: https://github.com/STEllAR-GROUP/hpx/issues/5163

2.10. Releases
• Issue #5161 - Bump required CMake version to 3.17
• Issue #5143 - Searching for HPX-Application to generate work on multiple Nodes

Closed pull requests

• PR #5438 - Delete datapar/foreach_tests.hpp
• PR #5437 - Add back explicit -pthread flags when available
• PR #5435 - This adds support for systems that assume all types are bitwise serializable by default
• PR #5434 - Update CUDA polling logging to be more verbose
• PR #5433 - Fix when_all_sender connect for references
• PR #5432 - Add deprecation warnings for v1.8
• PR #5431 - Rename the new P0443/P2300 executor to thread_pool_scheduler
• PR #5430 - Revert “Adding the missing defined for HPX_HAVE_DEPRECATED_WARNINGS”
• PR #5427 - Removing unneeded typedef
• PR #5426 - Adding more concept checks for sender/receiver algorithms
• PR #5425 - Adding the missing defined for HPX_HAVE_DEPRECATED_WARNINGS
• PR #5424 - Disable VC in final docker image created in CI
• PR #5422 - Adding execution::experimental::bulk algorithm
• PR #5420 - Update logic to find threading library
• PR #5418 - Reduce max size and number of files in ccache cache
• PR #5417 - Final release notes for 1.7.0
• PR #5416 - Adapt uninitialized_value_construct and uninitialized_value_construct_n to C++ 20
• PR #5415 - Adapt uninitialized_default_construct and uninitialized_default_construct_n to C++ 20
• PR #5414 - Improve integration of futures and senders

285 https://github.com/STEllAR-GROUP/hpx/issues/5161
286 https://github.com/STEllAR-GROUP/hpx/issue/5143
287 https://github.com/STEllAR-GROUP/hpx/pull/5438
288 https://github.com/STEllAR-GROUP/hpx/pull/5437
289 https://github.com/STEllAR-GROUP/hpx/pull/5435
290 https://github.com/STEllAR-GROUP/hpx/pull/5434
291 https://github.com/STEllAR-GROUP/hpx/pull/5433
292 https://github.com/STEllAR-GROUP/hpx/pull/5432
293 https://github.com/STEllAR-GROUP/hpx/pull/5431
294 https://github.com/STEllAR-GROUP/hpx/pull/5430
295 https://github.com/STEllAR-GROUP/hpx/pull/5427
296 https://github.com/STEllAR-GROUP/hpx/pull/5426
297 https://github.com/STEllAR-GROUP/hpx/pull/5425
298 https://github.com/STEllAR-GROUP/hpx/pull/5424
299 https://github.com/STEllAR-GROUP/hpx/pull/5422
300 https://github.com/STEllAR-GROUP/hpx/pull/5420
301 https://github.com/STEllAR-GROUP/hpx/pull/5418
302 https://github.com/STEllAR-GROUP/hpx/pull/5417
303 https://github.com/STEllAR-GROUP/hpx/pull/5416
304 https://github.com/STEllAR-GROUP/hpx/pull/5415
305 https://github.com/STEllAR-GROUP/hpx/pull/5414
- PR #5413 - Fixing sender/receiver code base to compile with MSVC
- PR #5407 - Handle exceptions thrown during initialization of parcel handler
- PR #5406 - Simplify dispatching to annotation handlers
- PR #5405 - Fetch Asio automatically in perftests CI
- PR #5403 - Create generic executor that adds annotations to any other executor
- PR #5402 - Adapt \texttt{uninitialized\_fill} and \texttt{uninitialized\_fill\_n} to C++ 20
- PR #5401 - Modernize a variety of facilities related to parallel algorithms
- PR #5400 - Fetch Asio automatically in perftests CI
- PR #5399 - Rename leftover \texttt{tag\_fallback\_invoke} to \texttt{tag\_fallback\_dispatch}
- PR #5398 - Improve logging in AGAS symbol namespace
- PR #5396 - Introduce compatibility layer for collective operations
- PR #5394 - Enable OTF2 in APEX CI configuration
- PR #5393 - Update APEX tag
- PR #5392 - Fixing wrong usage of \texttt{std::forward}
- PR #5391 - Fix forwarding in transform\_receiver constructor
- PR #5390 - Make sure shared priority scheduler steals tasks on the current NUMA domain when (core) stealing is enabled
- PR #5389 - Adapt \texttt{uninitialized\_move} and \texttt{uninitialized\_move\_n} to C++ 20
- PR #5388 - Fixing \texttt{gather\_there} for used with lvalue reference argument
- PR #5387 - Extend thread state logging and change default stealing parameters
- PR #5386 - Attempt to fix the startup hang with nodes > 32
- PR #5384 - Remove HPX 1.5.0 deprecations
- PR #5382 - Prefer installed Asio before considering FetchContent
- PR #5379 - Allow using pre-downloaded (not installed) versions of Asio and/or Apex

\footnotesize

306 https://github.com/STEllAR-GROUP/hpx/pull/5413
307 https://github.com/STEllAR-GROUP/hpx/pull/5407
308 https://github.com/STEllAR-GROUP/hpx/pull/5406
309 https://github.com/STEllAR-GROUP/hpx/pull/5405
310 https://github.com/STEllAR-GROUP/hpx/pull/5403
311 https://github.com/STEllAR-GROUP/hpx/pull/5402
312 https://github.com/STEllAR-GROUP/hpx/pull/5401
313 https://github.com/STEllAR-GROUP/hpx/pull/5400
314 https://github.com/STEllAR-GROUP/hpx/pull/5399
315 https://github.com/STEllAR-GROUP/hpx/pull/5398
316 https://github.com/STEllAR-GROUP/hpx/pull/5396
317 https://github.com/STEllAR-GROUP/hpx/pull/5394
318 https://github.com/STEllAR-GROUP/hpx/pull/5393
319 https://github.com/STEllAR-GROUP/hpx/pull/5392
320 https://github.com/STEllAR-GROUP/hpx/pull/5391
321 https://github.com/STEllAR-GROUP/hpx/pull/5390
322 https://github.com/STEllAR-GROUP/hpx/pull/5389
323 https://github.com/STEllAR-GROUP/hpx/pull/5388
324 https://github.com/STEllAR-GROUP/hpx/pull/5387
325 https://github.com/STEllAR-GROUP/hpx/pull/5386
326 https://github.com/STEllAR-GROUP/hpx/pull/5384
327 https://github.com/STEllAR-GROUP/hpx/pull/5382
328 https://github.com/STEllAR-GROUP/hpx/pull/5379
• PR #5376329 - Remove unnecessary explicit listing of library modules.rst files in CMakeLists.txt
• PR #5375330 - Slight performance improvement for hpx::copy and hpx::move et.al.
• PR #5374331 - Remove unnecessary moves from future sender implementations
• PR #5373332 - More changes to clang-cuda Jenkins configuration
• PR #5372333 - Slight improvements to min/max/minmax_element algorithms
• PR #5371334 - Adapt uninitialized_copy and uninitialized_copy_n to C++ 20
• PR #5370335 - Decay types in just_sender value_types to match stored types
• PR #5367336 - Disable pkgconfig by default again on macOS
• PR #5365337 - Use ccache for Jenkins builds on Piz Daint
• PR #5363338 - Update cudatoolkit module name in clang-cuda Jenkins configuration
• PR #5362339 - Adding channelCommunicator
• PR #5361340 - Fix compilation with MPI enabled
• PR #5360341 - Update APEX and asio tags
• PR #5359342 - Fix check for pu-step in single-core case
• PR #5357343 - Making sure collective operations can be reused by preallocating communicator
• PR #5356344 - Update API documentation
• PR #5355345 - Make the sequencedExecutor processing_units_count member function const
• PR #5354346 - Making sure default_stack_size is defined whenever declared
• PR #5353347 - Add CUDA timestamp support to HPX Hardware Clock
• PR #5352348 - Adding missing includes
• PR #5351349 - Adding enableLogging/disableLogging API functions
• PR #5350350 - Adapt lexicographical_compare to C++20
• PR #5349351 - Update minimum boost version needed on the docs

329 https://github.com/STEllAR-GROUP/hpx/pull/5376
330 https://github.com/STEllAR-GROUP/hpx/pull/5375
331 https://github.com/STEllAR-GROUP/hpx/pull/5374
332 https://github.com/STEllAR-GROUP/hpx/pull/5373
333 https://github.com/STEllAR-GROUP/hpx/pull/5372
334 https://github.com/STEllAR-GROUP/hpx/pull/5371
335 https://github.com/STEllAR-GROUP/hpx/pull/5370
336 https://github.com/STEllAR-GROUP/hpx/pull/5367
337 https://github.com/STEllAR-GROUP/hpx/pull/5365
338 https://github.com/STEllAR-GROUP/hpx/pull/5363
339 https://github.com/STEllAR-GROUP/hpx/pull/5362
340 https://github.com/STEllAR-GROUP/hpx/pull/5361
341 https://github.com/STEllAR-GROUP/hpx/pull/5360
342 https://github.com/STEllAR-GROUP/hpx/pull/5359
343 https://github.com/STEllAR-GROUP/hpx/pull/5357
344 https://github.com/STEllAR-GROUP/hpx/pull/5356
345 https://github.com/STEllAR-GROUP/hpx/pull/5355
346 https://github.com/STEllAR-GROUP/hpx/pull/5354
347 https://github.com/STEllAR-GROUP/hpx/pull/5353
348 https://github.com/STEllAR-GROUP/hpx/pull/5352
349 https://github.com/STEllAR-GROUP/hpx/pull/5351
350 https://github.com/STEllAR-GROUP/hpx/pull/5350
351 https://github.com/STEllAR-GROUP/hpx/pull/5349
• PR #5348\textsuperscript{352} - Rename \texttt{tag\_invoke} and related facilities to \texttt{tag\_dispatch}
• PR #5347\textsuperscript{353} - Remove \texttt{make\_prefix} for executor properties
• PR #5346\textsuperscript{354} - Remove and disable compatibility options for 1.7.0
• PR #5343\textsuperscript{355} - Fix \texttt{timed\_executor} static cast conversion
• PR #5342\textsuperscript{356} - Refactor CUDA event polling
• PR #5341\textsuperscript{357} - Adding \texttt{make\_with\_annotation} and \texttt{get\_annotation} properties
• PR #5339\textsuperscript{358} - Making sure \texttt{hpx::util::hardware::timestamp()} is always defined
• PR #5336\textsuperscript{359} - Fixing \texttt{timed\_executor} specializations of customization points
• PR #5335\textsuperscript{360} - Make \texttt{partial\_algorithm} work with any number of arguments
• PR #5334\textsuperscript{361} - Follow up \texttt{iter\_sent} include on #5225
• PR #5332\textsuperscript{362} - Simplify \texttt{tag\_invoke} and friends
• PR #5331\textsuperscript{363} - More work on cleaning up executor CPOs
• PR #5330\textsuperscript{364} - Add option to disable pkgconfig generation
• PR #5328\textsuperscript{365} - Adapt data parallel support using std-simd
• PR #5327\textsuperscript{366} - Fix missing \texttt{ifdef HPX\_SMT\_PAUSE}
• PR #5326\textsuperscript{367} - Adding \texttt{resize()} to \texttt{serialize\_buffer} allowing to shrink its size
• PR #5324\textsuperscript{368} - Add get member functions to \texttt{async\_rw\_mutex} proxy objects for explicitly getting the wrapped value
• PR #5323\textsuperscript{369} - Add \texttt{keep\_future} algorithm
• PR #5322\textsuperscript{370} - Replace executor customization point implementations with \texttt{tag\_invoke}
• PR #5321\textsuperscript{371} - Separated segmented algorithms for reduce
• PR #5320\textsuperscript{372} - Fix \texttt{is\_sender} trait and other small fixes to p0443 traits
• PR #5319\textsuperscript{373} - gcc 11.1 c++20 build fixes
• PR #5318\textsuperscript{374} - Make hipblas dependency optional as not always available

\textsuperscript{352} https://github.com/STEllAR-GROUP/hpx/pull/5348
\textsuperscript{353} https://github.com/STEllAR-GROUP/hpx/pull/5347
\textsuperscript{354} https://github.com/STEllAR-GROUP/hpx/pull/5346
\textsuperscript{355} https://github.com/STEllAR-GROUP/hpx/pull/5343
\textsuperscript{356} https://github.com/STEllAR-GROUP/hpx/pull/5342
\textsuperscript{357} https://github.com/STEllAR-GROUP/hpx/pull/5341
\textsuperscript{358} https://github.com/STEllAR-GROUP/hpx/pull/5339
\textsuperscript{359} https://github.com/STEllAR-GROUP/hpx/pull/5338
\textsuperscript{360} https://github.com/STEllAR-GROUP/hpx/pull/5335
\textsuperscript{361} https://github.com/STEllAR-GROUP/hpx/pull/5334
\textsuperscript{362} https://github.com/STEllAR-GROUP/hpx/pull/5332
\textsuperscript{363} https://github.com/STEllAR-GROUP/hpx/pull/5331
\textsuperscript{364} https://github.com/STEllAR-GROUP/hpx/pull/5330
\textsuperscript{365} https://github.com/STEllAR-GROUP/hpx/pull/5328
\textsuperscript{366} https://github.com/STEllAR-GROUP/hpx/pull/5327
\textsuperscript{367} https://github.com/STEllAR-GROUP/hpx/pull/5326
\textsuperscript{368} https://github.com/STEllAR-GROUP/hpx/pull/5324
\textsuperscript{369} https://github.com/STEllAR-GROUP/hpx/pull/5323
\textsuperscript{370} https://github.com/STEllAR-GROUP/hpx/pull/5322
\textsuperscript{371} https://github.com/STEllAR-GROUP/hpx/pull/5321
\textsuperscript{372} https://github.com/STEllAR-GROUP/hpx/pull/5320
\textsuperscript{373} https://github.com/STEllAR-GROUP/hpx/pull/5319
\textsuperscript{374} https://github.com/STEllAR-GROUP/hpx/pull/5318
• PR #5316\(^{375}\) - Attempt to fix checking for libatomic
• PR #5315\(^{376}\) - Add explicit keyword to fixture constructor
• PR #5314\(^{377}\) - Fix a race condition in async mpi affecting limiting executor
• PR #5312\(^{378}\) - Use local runtime and local headers in local-only modules and tests
• PR #5311\(^{379}\) - Add GCC 11 builder to jenkins
• PR #5310\(^{380}\) - Adding hpx::execution::experimental::task_group
• PR #5309\(^{381}\) - Separate datapar
• PR #5308\(^{382}\) - Separate segmented algorithms for find, find_if, find_if_not
• PR #5307\(^{383}\) - Separate segmented algorithms for fill and generate
• PR #5304\(^{384}\) - Fix compilation of sender CPOs with nvcc
• PR #5303\(^{385}\) - Remove PRIVATE flag that was propagated into the LANGUAGES
• PR #5298\(^{386}\) - Separate datapar
• PR #5297\(^{387}\) - Specify exact cmake and ninja versions when loading them in jenkins jobs
• PR #5295\(^{388}\) - Update clang-newest configuration to use clang 12 and Boost 1.76.0
• PR #5293\(^{389}\) - Fix Clang 11 cuda_future test bug
• PR #5292\(^{390}\) - Add async_rw_mutex based on senders
• PR #5291\(^{391}\) - “Fix” termination detection
• PR #5290\(^{392}\) - Fixed source file line statements in examples documentation
• PR #5289\(^{393}\) - Allow splitting of futures holding std::tuple
• PR #5288\(^{394}\) - Move algorithms to tag_fallback_invoke
• PR #5287\(^{395}\) - Move algorithms to tag_fallback_invoke
• PR #5285\(^{396}\) - Fix clang-format failure on master
• PR #5284\(^{397}\) - Replacing util::function_nonser on std::function in hpx_init
- PR #5282\(^{398}\) - Update Boost for daint 20.11 after update
- PR #5281\(^{399}\) - Fix Segmentation fault on foreach_datapar_zipiter
- PR #5280\(^{400}\) - Avoid modulo by zero in counting_iterator test
- PR #5279\(^{401}\) - Fix more GCC 10 deprecation warnings
- PR #5277\(^{402}\) - Small fixes and improvements to CUDA/mpi polling
- PR #5276\(^{403}\) - Fix typo in docs
- PR #5274\(^{404}\) - More P1897 algorithms
- PR #5273\(^{405}\) - Retry CDash submissions on failure
- PR #5272\(^{406}\) - Fix bogus deprecation warnings with GCC 10
- PR #5271\(^{407}\) - Correcting target ids for symbol_namespace::iterate
- PR #5268\(^{408}\) - Adding generic require, require_concept, and query properties
- PR #5267\(^{409}\) - Support annotations in hpx::transform_reduce
- PR #5266\(^{410}\) - Making late command line options available for local runtime
- PR #5265\(^{411}\) - Leverage no_unique_address for member_pack
- PR #5264\(^{412}\) - Adopt format in more places
- PR #5262\(^{413}\) - Install HPX in Rostam Jenkins jobs
- PR #5261\(^{414}\) - Limit Rostam Jenkins jobs to marvin partition temporarily
- PR #5260\(^{415}\) - Separate segmented algorithms for transform_reduce
- PR #5259\(^{416}\) - Making sure late command line options are recognized as configuration options
- PR #5258\(^{417}\) - Allow for HPX algorithms being invoked with std execution policies
- PR #5256\(^{418}\) - Separate segmented algorithms for transform
- PR #5255\(^{419}\) - Future/sender adapters
- PR #5254\(^{420}\) - Fixing datapar

[398] https://github.com/STEllAR-GROUP/hpx/pull/5282
[399] https://github.com/STEllAR-GROUP/hpx/pull/5281
[400] https://github.com/STEllAR-GROUP/hpx/pull/5280
[401] https://github.com/STEllAR-GROUP/hpx/pull/5279
[402] https://github.com/STEllAR-GROUP/hpx/pull/5278
[403] https://github.com/STEllAR-GROUP/hpx/pull/5277
[404] https://github.com/STEllAR-GROUP/hpx/pull/5276
[405] https://github.com/STEllAR-GROUP/hpx/pull/5275
[406] https://github.com/STEllAR-GROUP/hpx/pull/5274
[407] https://github.com/STEllAR-GROUP/hpx/pull/5273
[408] https://github.com/STEllAR-GROUP/hpx/pull/5272
[409] https://github.com/STEllAR-GROUP/hpx/pull/5271
[410] https://github.com/STEllAR-GROUP/hpx/pull/5268
[411] https://github.com/STEllAR-GROUP/hpx/pull/5267
[412] https://github.com/STEllAR-GROUP/hpx/pull/5266
[413] https://github.com/STEllAR-GROUP/hpx/pull/5265
[414] https://github.com/STEllAR-GROUP/hpx/pull/5264
[415] https://github.com/STEllAR-GROUP/hpx/pull/5263
[416] https://github.com/STEllAR-GROUP/hpx/pull/5262
[417] https://github.com/STEllAR-GROUP/hpx/pull/5261
[418] https://github.com/STEllAR-GROUP/hpx/pull/5260
[419] https://github.com/STEllAR-GROUP/hpx/pull/5259
[420] https://github.com/STEllAR-GROUP/hpx/pull/5258
[421] https://github.com/STEllAR-GROUP/hpx/pull/5257
[422] https://github.com/STEllAR-GROUP/hpx/pull/5256
[423] https://github.com/STEllAR-GROUP/hpx/pull/5255
[424] https://github.com/STEllAR-GROUP/hpx/pull/5254
• PR #5253 - Add utility to format ranges
• PR #5254 - Remove uses of Boost.Bimap
• PR #5251 - Banish <iostream> from library headers
• PR #5250 - Try fixing vc circle ci
• PR #5249 - Adding missing header
• PR #5248 - Use old Piz Daint modules after upgrade
• PR #5247 - Significantly speedup simple for_each, for_loop, and transform
• PR #5245 - P1897 operator| overloads
• PR #5244 - P1897 when_all
• PR #5243 - Make sure HPX_DEBUG is set based on HPX’s build type, not consuming project’s build type
• PR #5242 - Moving last files unrelated to parcel layer to modules
• PR #5240 - change namespace for transform_loop.hpp
• PR #5238 - Make sure annotations are used in the binary transform
• PR #5237 - Add P1897 just, just_on, and on algorithms
• PR #5236 - Add an example demonstrating the use of the invoke_function_action facility
• PR #5235 - Attempting to fix datapar compilation issues
• PR #5234 - Fix small typo in --hpx:local option description
• PR #5233 - Only find Boost.Iostreams if required for plugins
• PR #5231 - Sort printed config options
• PR #5230 - Fix C++20 replace algo adaptation misses
• PR #5229 - Remove leftover Boost include from sync_wait.hpp
• PR #5228 - Print module name only if it has custom configuration settings
• PR #5227 - Update .codespell_whitelist

https://github.com/STEllAR-GROUP/hpx/pull/5253
https://github.com/STEllAR-GROUP/hpx/pull/5252
https://github.com/STEllAR-GROUP/hpx/pull/5251
https://github.com/STEllAR-GROUP/hpx/pull/5250
https://github.com/STEllAR-GROUP/hpx/pull/5249
https://github.com/STEllAR-GROUP/hpx/pull/5248
https://github.com/STEllAR-GROUP/hpx/pull/5247
https://github.com/STEllAR-GROUP/hpx/pull/5245
https://github.com/STEllAR-GROUP/hpx/pull/5244
https://github.com/STEllAR-GROUP/hpx/pull/5243
https://github.com/STEllAR-GROUP/hpx/pull/5242
https://github.com/STEllAR-GROUP/hpx/pull/5240
https://github.com/STEllAR-GROUP/hpx/pull/5238
https://github.com/STEllAR-GROUP/hpx/pull/5237
https://github.com/STEllAR-GROUP/hpx/pull/5236
https://github.com/STEllAR-GROUP/hpx/pull/5235
https://github.com/STEllAR-GROUP/hpx/pull/5234
https://github.com/STEllAR-GROUP/hpx/pull/5233
https://github.com/STEllAR-GROUP/hpx/pull/5231
https://github.com/STEllAR-GROUP/hpx/pull/5230
https://github.com/STEllAR-GROUP/hpx/pull/5229
https://github.com/STEllAR-GROUP/hpx/pull/5228
https://github.com/STEllAR-GROUP/hpx/pull/5227
• PR #5226 - Use new docker image in all CircleCI steps
• PR #5225 - Adapt reverse to C++20
• PR #5224 - Separate segmented algorithms for none_of, any_of and all_of
• PR #5223 - Fixing build system for ittnotify
• PR #5221 - Moving LCO related files to modules
• PR #5220 - Separate segmented algorithms for count and count_if
• PR #5218 - Separate segmented algorithms for adjacent_find
• PR #5217 - Add a HIP github action
• PR #5215 - Update ROCm to 4.0.1 on Rostam
• PR #5214 - Fix clang-format error in sender.hpp
• PR #5213 - Removing ESSENTIAL option to the doc example
• PR #5212 - Separate segmented algorithms for for_each_n
• PR #5211 - Minor adapted algos fixes
• PR #5210 - Fixing is_invocable deprecation warnings
• PR #5209 - Moving more files into modules (actions, components, init_runtime, etc.)
• PR #5208 - Add examples and explanation on when tag_fallback/priority are useful
• PR #5207 - Always define HPX_COMPUTE_HOST_CODE for host code
• PR #5206 - Add formatting exceptions for libhpx to create_module_skeleton.py
• PR #5205 - Moving all distribution policies into modules
• PR #5203 - Move copy algorithms to tag_fallback_invoke
• PR #5202 - Make HPX_WITH_PSEUDO_DEPENDENCIES a cache variable
• PR #5201 - Replaced tag_invoke with tag_fallback_invoke for adjacent_find algorithm
• PR #5200 - Moving files to (distributed) runtime module
- PR #5199 - Update ICC module name on Piz Daint Jenkins configuration
- PR #5198 - Add doxygen documentation for thread_schedule_hint
- PR #5197 - Attempt to fix compilation of context implementations with unity build enabled
- PR #5196 - Re-enable component tests
- PR #5195 - Moving files related to colocation logic
- PR #5194 - Another attempt at fixing the Fedora 35 problem
- PR #5193 - Components module
- PR #5192 - Adapt replace(_if) to C++20
- PR #5190 - Set compatibility headers by default to on
- PR #5188 - Bump Boost minimum version to 1.71.0
- PR #5187 - Force CMake to set the -std=c++XX flag
- PR #5186 - Remove message to print .cu extension whenever .cu files are encountered
- PR #5185 - Remove some minor unnecessary CMake options
- PR #5184 - Remove some leftover HPX_WITH_*_SCHEDULER uses
- PR #5183 - Remove dependency on boost/iterators/iterator_categories.hpp
- PR #5182 - Fixing Fedora 35 for Power architectures
- PR #5181 - Bump version number and tag post 1.6.0 release
- PR #5180 - Fix httsv2 tests linking
- PR #5179 - Make sure --hpx:local command line option is respected with networking is off but distributed runtime is on
- PR #5177 - Remove module cmake options
- PR #5176 - Starting to separate segmented algorithms: for_each
- PR #5174 - Don’t run segmented algorithms twice on CircleCI
- PR #5173 - Fetching APEX using cmake FetchContent
- PR #5172 - Add separate local-only entry point
- PR #5171 - Remove HPX\_WITH\_THREAD\_SCHEDULERS CMake option
- PR #5170 - Add HPX\_WITH\_PRECOMPILED\_HEADERS option
- PR #5166 - Moving some action tests to modules
- PR #5165 - Require cmake 3.17
- PR #5164 - Move thread\_pool\_suspension\_helper files to small utility module
- PR #5160 - Adding checks ensuring modules are not cross-referenced from other module categories
- PR #5158 - Replace boost::asio with standalone asio
- PR #5155 - Allow logging when distributed runtime is off
- PR #5153 - Components module
- PR #5152 - Move more files to performance counter module
- PR #5150 - Adapt remove\_copy\_(\_if) to C++20
- PR #5144 - AGAS module
- PR #5125 - Adapt remove and remove\_if to C++20
- PR #5117 - Attempt to fix segfaults assumed to be caused by future\_data instances going out of scope.
- PR #5099 - Allow mixing debug and release builds
- PR #5092 - Replace spirit.qi with x3
- PR #5053 - Add P0443r14 executor and a a few P1897 algorithms
- PR #5044 - Add performance test in jenkins and reports

https://github.com/STEllAR-GROUP/hpx/pull/5172
https://github.com/STEllAR-GROUP/hpx/pull/5171
https://github.com/STEllAR-GROUP/hpx/pull/5170
https://github.com/STEllAR-GROUP/hpx/pull/5166
https://github.com/STEllAR-GROUP/hpx/pull/5165
https://github.com/STEllAR-GROUP/hpx/pull/5164
https://github.com/STEllAR-GROUP/hpx/pull/5160
https://github.com/STEllAR-GROUP/hpx/pull/5158
https://github.com/STEllAR-GROUP/hpx/pull/5155
https://github.com/STEllAR-GROUP/hpx/pull/5153
https://github.com/STEllAR-GROUP/hpx/pull/5152
https://github.com/STEllAR-GROUP/hpx/pull/5150
https://github.com/STEllAR-GROUP/hpx/pull/5144
https://github.com/STEllAR-GROUP/hpx/pull/5125
https://github.com/STEllAR-GROUP/hpx/pull/5117
https://github.com/STEllAR-GROUP/hpx/pull/5099
https://github.com/STEllAR-GROUP/hpx/pull/5092
https://github.com/STEllAR-GROUP/hpx/pull/5053
https://github.com/STEllAR-GROUP/hpx/pull/5044
General changes

This release continues the focus on C++20 conformance with multiple new algorithms adapted to be C++20 confor-
mant 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\(^509\). 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,

\(^509\) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2220r0.pdf
HPX\_WITH\_POOL\_EXECUTOR\_COMPATIBILITY. Unless noted here, the above functionalities do not come 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\(^{510}\) - runtime\_support\_hpp does not work with newer cling
- Issue #5147\(^{511}\) - Wrong results with parallel reduce
- Issue #5129\(^{512}\) - Missing specialization for std::hash<hpx::thread::id>
- Issue #5126\(^{513}\) - Use std::string for task annotations
- Issue #5115\(^{514}\) - Don’t expect hwloc to always report Cores
- Issue #5113\(^{515}\) - Handle threadmanager exceptions during startup
- Issue #5112\(^{516}\) - libatomic problems causing unexpected fails
- Issue #5089\(^{517}\) - Remove non-BSL files
- Issue #5088\(^{518}\) - Unwrapping problem
- Issue #5087\(^{519}\) - Remove hpxMP support
- Issue #5077\(^{520}\) - PAPI counters are not accessible when HPX is installed
- Issue #5075\(^{521}\) - Make the structs in all iter\_sent\_hpp lower case
- Issue #5067\(^{522}\) - Bug string\_util/split\_hpp
- Issue #5049\(^{523}\) - Change back the hipcc jenkins config to the fury partition on rostam
- Issue #5038\(^{524}\) - Not all examples link in the latest HPX master
- Issue #5035\(^{525}\) - Build with HPX\_WITH\_EXAMPLES fails
- Issue #5019\(^{526}\) - Broken help string for hpx
- Issue #5016\(^{527}\) - hpx::parallel::fill fails compiling

\(^{510}\)https://github.com/STEllAR-GROUP/hpx/issues/5148
\(^{511}\)https://github.com/STEllAR-GROUP/hpx/issues/5147
\(^{512}\)https://github.com/STEllAR-GROUP/hpx/issues/5129
\(^{513}\)https://github.com/STEllAR-GROUP/hpx/issues/5126
\(^{514}\)https://github.com/STEllAR-GROUP/hpx/issues/5115
\(^{515}\)https://github.com/STEllAR-GROUP/hpx/issues/5113
\(^{516}\)https://github.com/STEllAR-GROUP/hpx/issues/5112
\(^{517}\)https://github.com/STEllAR-GROUP/hpx/issues/5088
\(^{518}\)https://github.com/STEllAR-GROUP/hpx/issues/5089
\(^{519}\)https://github.com/STEllAR-GROUP/hpx/issues/5087
\(^{520}\)https://github.com/STEllAR-GROUP/hpx/issues/5077
\(^{521}\)https://github.com/STEllAR-GROUP/hpx/issues/5075
\(^{522}\)https://github.com/STEllAR-GROUP/hpx/issues/5067
\(^{523}\)https://github.com/STEllAR-GROUP/hpx/issues/5049
\(^{524}\)https://github.com/STEllAR-GROUP/hpx/issues/5038
\(^{525}\)https://github.com/STEllAR-GROUP/hpx/issues/5035
\(^{526}\)https://github.com/STEllAR-GROUP/hpx/issues/5019
\(^{527}\)https://github.com/STEllAR-GROUP/hpx/issues/5016
• Issue #5014 - Rename all .cc to .cpp and .hh to .hpp
• Issue #4988 - MPI is not finalized if running with only one locality
• Issue #4978 - Change feature test macros to expand to zero/one
• Issue #4949 - Crash when not enabling TCP parcelport
• Issue #4933 - Improve test coverage for unused variable warnings etc.
• Issue #4878 - HPX mpi async might call MPI_FINALIZE before app calls it
• Issue #4127 - Local runtime entry-points

Closed pull requests

• PR #5178 - Fix parallel remove/remove_copy/transform namespace references in docs
• PR #5169 - Attempt to get Piz Daint jenkins setup running after maintenance
• PR #5168 - Remove include of itself
• PR #5167 - Fixing deprecation warnings that slipped through the net
• PR #5159 - Update APEX tag to 2.3.1
• PR #5154 - Splitting unit tests on circleci to avoid timeouts
• PR #5151 - Use C++20 on clang-newest Jenkins CI configuration
• PR #5149 - Rename 'module' symbols to avoid keyword conflict
• PR #5147 - Adjust handling of CUDA/HIP options in CMake
• PR #5145 - Store annotated_function annotations as std::strings
• PR #5140 - Scheduler mode
• PR #5139 - Fix path problem in pre-commit hook, add summary commit line
• PR #5138 - Add program options variable map to resource partitioner init
• PR #5137 - Remove the use of boost::throw_exception
• PR #5136 - Make sure codespell checks run on CircleCI
• PR #5132\(^{550}\) - Fixing spelling errors
• PR #5131\(^{551}\) - Mark `counting_iterator` member functions as `HPX_HOST_DEVICE`
• PR #5130\(^{552}\) - Adding specialization for `std::hash<hpx::thread::id>`
• PR #5128\(^{553}\) - Fixing environment handling for FreeBSD
• PR #5127\(^{554}\) - Fix typo in fibonacci documentation
• PR #5123\(^{555}\) - Reduce vector sizes in partial sort benchmarks when running in debug mode
• PR #5122\(^{556}\) - Making sure exceptions during runtime initialization are correctly reported
• PR #5121\(^{557}\) - Working around hwloc limitation on certain platforms
• PR #5120\(^{558}\) - Fixing compatibility warnings in `hpx::transform` implementation
• PR #5119\(^{559}\) - Use `sequential_find` and friends from separate detail header
• PR #5116\(^{560}\) - Fix compilation with timer pool off
• PR #5114\(^{561}\) - Fix 5112 - make sure libatomic is used when needed
• PR #5109\(^{562}\) - Remove default runtime mode argument from init overload, again
• PR #5108\(^{563}\) - Refactor `iter_sent.hpp` to make structs lowercase
• PR #5107\(^{564}\) - Relax dataflow internals
• PR #5106\(^{565}\) - Change initialization of property CPOs to satisfy older nvcc versions
• PR #5104\(^{566}\) - Fix regeneration of two files that trigger unnecessary rebuilds
• PR #5103\(^{567}\) - Remove default runtime mode argument from start/init overloads
• PR #5102\(^{568}\) - Untie deprecated thread enums from the CMake option
• PR #5101\(^{569}\) - Update APEX tag for 1.6.0
• PR #5100\(^{570}\) - Bump minimum required Boost version to 1.66 and update CI configurations
• PR #5098\(^{571}\) - Minor fixes to public API listing
• PR #5097\(^{572}\) - Remove hpxMP support

\(^{550}\) https://github.com/STEllAR-GROUP/hpx/pull/5132
\(^{551}\) https://github.com/STEllAR-GROUP/hpx/pull/5131
\(^{552}\) https://github.com/STEllAR-GROUP/hpx/pull/5130
\(^{553}\) https://github.com/STEllAR-GROUP/hpx/pull/5128
\(^{554}\) https://github.com/STEllAR-GROUP/hpx/pull/5127
\(^{555}\) https://github.com/STEllAR-GROUP/hpx/pull/5123
\(^{556}\) https://github.com/STEllAR-GROUP/hpx/pull/5122
\(^{557}\) https://github.com/STEllAR-GROUP/hpx/pull/5121
\(^{558}\) https://github.com/STEllAR-GROUP/hpx/pull/5120
\(^{559}\) https://github.com/STEllAR-GROUP/hpx/pull/5119
\(^{560}\) https://github.com/STEllAR-GROUP/hpx/pull/5116
\(^{561}\) https://github.com/STEllAR-GROUP/hpx/pull/5114
\(^{562}\) https://github.com/STEllAR-GROUP/hpx/pull/5109
\(^{563}\) https://github.com/STEllAR-GROUP/hpx/pull/5108
\(^{564}\) https://github.com/STEllAR-GROUP/hpx/pull/5107
\(^{565}\) https://github.com/STEllAR-GROUP/hpx/pull/5106
\(^{566}\) https://github.com/STEllAR-GROUP/hpx/pull/5104
\(^{567}\) https://github.com/STEllAR-GROUP/hpx/pull/5103
\(^{568}\) https://github.com/STEllAR-GROUP/hpx/pull/5102
\(^{569}\) https://github.com/STEllAR-GROUP/hpx/pull/5101
\(^{570}\) https://github.com/STEllAR-GROUP/hpx/pull/5100
\(^{571}\) https://github.com/STEllAR-GROUP/hpx/pull/5098
\(^{572}\) https://github.com/STEllAR-GROUP/hpx/pull/5097

2.10. Releases
• PR #5096 - Remove fractals examples
• PR #5095 - Use all AMD nodes again on rostam
• PR #5094 - Attempt to remove macOS workaround for GH actions environment
• PR #5093 - Remove verbs parcelport
• PR #5091 - Avoid moving from lvalues
• PR #5090 - Adopt C++20 std::endian
• PR #5085 - Update daint CI to use Boost 1.75.0
• PR #5084 - Disable compatibility options for 1.6.0 release
• PR #5083 - Remove duplicated call to the limiting_executor in future_overhead test
• PR #5079 - Add checks to make sure that MPI/CUDA polling is enabled/not disabled too early
• PR #5078 - Add install lib directory to list of component search paths
• PR #5076 - Fix a typo in the jenkins clang-newest cmake config
• PR #5074 - Fixing warnings generated by MSVC
• PR #5073 - Allow using noncopyable types with unwrapping
• PR #5072 - Fix is_convertible args in result_types
• PR #5071 - Fix unused parameters
• PR #5070 - Fix unused variables warnings in hipcc
• PR #5069 - Add support for sentinels to adjacent_find
• PR #5068 - Fix string split function
• PR #5066 - Adapt search to C++20 and Range TS
• PR #5065 - Fix hpx::range::adjacent_find doxygen function signatures
• PR #5064 - Refactor runtime configuration, command line handling, and resource partitioner
• PR #5063 - Limit the device code guards to the distributed parts of the future_overhead bench

https://github.com/STEllAR-GROUP/hpx/pull/5096
https://github.com/STEllAR-GROUP/hpx/pull/5095
https://github.com/STEllAR-GROUP/hpx/pull/5094
https://github.com/STEllAR-GROUP/hpx/pull/5093
https://github.com/STEllAR-GROUP/hpx/pull/5091
https://github.com/STEllAR-GROUP/hpx/pull/5090
https://github.com/STEllAR-GROUP/hpx/pull/5085
https://github.com/STEllAR-GROUP/hpx/pull/5084
https://github.com/STEllAR-GROUP/hpx/pull/5083
https://github.com/STEllAR-GROUP/hpx/pull/5079
https://github.com/STEllAR-GROUP/hpx/pull/5078
https://github.com/STEllAR-GROUP/hpx/pull/5076
https://github.com/STEllAR-GROUP/hpx/pull/5074
https://github.com/STEllAR-GROUP/hpx/pull/5073
https://github.com/STEllAR-GROUP/hpx/pull/5072
https://github.com/STEllAR-GROUP/hpx/pull/5071
https://github.com/STEllAR-GROUP/hpx/pull/5070
https://github.com/STEllAR-GROUP/hpx/pull/5069
https://github.com/STEllAR-GROUP/hpx/pull/5068
https://github.com/STEllAR-GROUP/hpx/pull/5066
https://github.com/STEllAR-GROUP/hpx/pull/5065
https://github.com/STEllAR-GROUP/hpx/pull/5064
https://github.com/STEllAR-GROUP/hpx/pull/5063
• PR #5061 - Remove hipcc guards in examples and tests
• PR #5060 - Fix deprecation warnings generated by msvc
• PR #5059 - Add warning about suspending/resuming the runtime in multi-locality scenarios
• PR #5057 - Fix unused variable warnings
• PR #5056 - Fix hpx::util::get
• PR #5055 - Remove hipcc guards
• PR #5054 - Fix typo
• PR #5051 - Adapt transform to C++20
• PR #5050 - Replace old init overloads in tests and examples
• PR #5048 - Limit jenkins hipcc to the reno node
• PR #5047 - Limit cuda jenkins run to nodes with exclusively Nvidia GPUs
• PR #5046 - Convert thread and future enums to class enums
• PR #5043 - Improve hpxrun.py for Phylanx
• PR #5042 - Add missing header to partial sort test
• PR #5041 - Adding Francisco Tapia’s implementation of partial_sort
• PR #5040 - Remove generated headers left behind from a previous configuration
• PR #5039 - Fix GCC 10 release builds
• PR #5037 - Add is_invocable typedefs to top-level hpx namespace and public API list
• PR #5036 - Deprecate hpx::util::decay in favor of std::decay
• PR #5034 - Use versioned container image on CircleCI
• PR #5032 - Implement P2220 properties module
• PR #5031 - Do codespell comparison only on files changed from common ancestor
• PR #5030 - Moving traits files to actions_base

https://github.com/STEllAR-GROUP/hpx/pull/5061
https://github.com/STEllAR-GROUP/hpx/pull/5060
https://github.com/STEllAR-GROUP/hpx/pull/5059
https://github.com/STEllAR-GROUP/hpx/pull/5057
https://github.com/STEllAR-GROUP/hpx/pull/5056
https://github.com/STEllAR-GROUP/hpx/pull/5055
https://github.com/STEllAR-GROUP/hpx/pull/5054
https://github.com/STEllAR-GROUP/hpx/pull/5051
https://github.com/STEllAR-GROUP/hpx/pull/5050
https://github.com/STEllAR-GROUP/hpx/pull/5048
https://github.com/STEllAR-GROUP/hpx/pull/5047
https://github.com/STEllAR-GROUP/hpx/pull/5046
https://github.com/STEllAR-GROUP/hpx/pull/5043
https://github.com/STEllAR-GROUP/hpx/pull/5042
https://github.com/STEllAR-GROUP/hpx/pull/5041
https://github.com/STEllAR-GROUP/hpx/pull/5040
https://github.com/STEllAR-GROUP/hpx/pull/5039
https://github.com/STEllAR-GROUP/hpx/pull/5037
https://github.com/STEllAR-GROUP/hpx/pull/5036
https://github.com/STEllAR-GROUP/hpx/pull/5034
https://github.com/STEllAR-GROUP/hpx/pull/5033
https://github.com/STEllAR-GROUP/hpx/pull/5032
https://github.com/STEllAR-GROUP/hpx/pull/5031

2.10. Releases
• PR #5030 - Add codespell version print in circleci
• PR #5029 - Work around problems in GitHub actions macOS builder
• PR #5028 - Moving move files to naming and naming_base
• PR #5027 - Lessen constraints on certain algorithm arguments
• PR #5025 - Adapt is_sorted and is_sorted_until to C++20
• PR #5024 - Moving naming_base to full modules
• PR #5023 - Remove C language from CMakeLists.txt
• PR #5021 - Warn about unused arguments given to add_hpx_module
• PR #5020 - Fixing help string
• PR #5018 - Update CSCS jenkins configuration to clang 11
• PR #5017 - Fixing broken backwards compatibility for hpx::parallel::fill
• PR #5015 - Detect if generated global header conflicts with explicitly listed module headers
• PR #5012 - Properly reset pointer tracking data in output_archive
• PR #5011 - Inspect command line tweaks
• PR #5010 - Creating AGAS module
• PR #5009 - Replace boost::system::error_code with std::error_code
• PR #5008 - Replace uses of boost::detail::spinlock
• PR #5007 - Bump minimal Boost version to 1.65.0
• PR #5006 - Adapt is_partitioned to C++20
• PR #5005 - Making sure reduce_by_key compiles again
• PR #5004 - Fixing template specializations that make extra archive data types unique across module boundaries
• PR #5003 - Relax dataflow argument constraints
• PR #5001 - Add <random> inspect check
• PR #4999 - Attempt to fix MacOS Github action error
• PR #4997 - Fix unused variable and typedef warnings
• PR #4996 - Adapt adjacent_find to C++20
• PR #4995 - Test all schedulers in cross_pool_injection test except shared_priority_queue_scheduler
• PR #4993 - Fix deprecation warnings
• PR #4991 - Avoid unnecessarily including entire modules
• PR #4990 - Fixing some warnings from HPX complaining about use of obsolete types
• PR #4989 - add a *destroy* trait for ParcelPort plugins
• PR #4986 - Remove serialization to functional module dependency
• PR #4985 - Compatibility header generation
• PR #4980 - Add ranges overloads to for_loop (and variants)
• PR #4979 - Actually enable unity builds on Jenkins
• PR #4976 - Cleaning up debug::print functionalities
• PR #4975 - Remove indirection layer in at_index_impl
• PR #4973 - Avoid warnings/errors for older gcc complaining about multi-line comments
• PR #4970 - Making set algorithms conform to C++20
• PR #4969 - Moving is_execution_policy and friends into namespace hpx
• PR #4968 - Enable deprecation warnings for 1.6.0 and move any functionality to hpx namespace
• PR #4967 - Define deprecation macros conditionally
• PR #4966 - Add clang-format and cmake-format version prints
• PR #4965 - Making is_heap and is_heap_until conforming to C++20
• PR #4964 - Adding parallel make_heap

642 https://github.com/STEllAR-GROUP/hpx/pull/4999
643 https://github.com/STEllAR-GROUP/hpx/pull/4997
644 https://github.com/STEllAR-GROUP/hpx/pull/4996
645 https://github.com/STEllAR-GROUP/hpx/pull/4995
646 https://github.com/STEllAR-GROUP/hpx/pull/4993
647 https://github.com/STEllAR-GROUP/hpx/pull/4991
648 https://github.com/STEllAR-GROUP/hpx/pull/4990
649 https://github.com/STEllAR-GROUP/hpx/pull/4989
650 https://github.com/STEllAR-GROUP/hpx/pull/4986
651 https://github.com/STEllAR-GROUP/hpx/pull/4985
652 https://github.com/STEllAR-GROUP/hpx/pull/4980
653 https://github.com/STEllAR-GROUP/hpx/pull/4979
654 https://github.com/STEllAR-GROUP/hpx/pull/4977
655 https://github.com/STEllAR-GROUP/hpx/pull/4976
656 https://github.com/STEllAR-GROUP/hpx/pull/4975
657 https://github.com/STEllAR-GROUP/hpx/pull/4973
658 https://github.com/STEllAR-GROUP/hpx/pull/4970
659 https://github.com/STEllAR-GROUP/hpx/pull/4969
660 https://github.com/STEllAR-GROUP/hpx/pull/4968
661 https://github.com/STEllAR-GROUP/hpx/pull/4967
662 https://github.com/STEllAR-GROUP/hpx/pull/4966
663 https://github.com/STEllAR-GROUP/hpx/pull/4965
664 https://github.com/STEllAR-GROUP/hpx/pull/4964
• PR #4962665 - Fix external timer function pointer exports
• PR #4960666 - Fixing folder names for module tests and examples
• PR #4959667 - Adding communications set
• PR #4958668 - Deprecate tuple and timing functionality in hpx::util
• PR #4957669 - Fixing unity build option for parcelports
• PR #4953670 - Fixing MSVC problems after recent restructurings
• PR #4952671 - Make parallel_executor use thread_pool_executor spawning mechanism
• PR #4948672 - Clean up old artifacts better and more aggressively on Jenkins
• PR #4947673 - Add HIP support for AMD GPUs
• PR #4945674 - Enable HPX_WITH_UNITY_BUILD option on one of the Jenkins configurations
• PR #4943675 - Move public hpx::parallel::execution functionality to hpx::execution
• PR #4938676 - Post release cleanup
• PR #4858677 - Extending resilience APIs to support distributed invocations
• PR #4744678 - Fork-join executor
• PR #4665679 - Implementing sender, receiver, and operation_state concepts in terms of P0443r13
• PR #4649680 - Split libhpx into multiple libraries
• PR #4642681 - Implementing operation_state concept in terms of P0443r13
• PR #4640682 - Implementing receiver concept in terms of P0443r13
• PR #4627683 - Sanitizer fixes

665 https://github.com/STEllAR-GROUP/hpx/pull/4962
666 https://github.com/STEllAR-GROUP/hpx/pull/4960
667 https://github.com/STEllAR-GROUP/hpx/pull/4959
668 https://github.com/STEllAR-GROUP/hpx/pull/4958
669 https://github.com/STEllAR-GROUP/hpx/pull/4957
670 https://github.com/STEllAR-GROUP/hpx/pull/4953
671 https://github.com/STEllAR-GROUP/hpx/pull/4952
672 https://github.com/STEllAR-GROUP/hpx/pull/4948
673 https://github.com/STEllAR-GROUP/hpx/pull/4947
674 https://github.com/STEllAR-GROUP/hpx/pull/4945
675 https://github.com/STEllAR-GROUP/hpx/pull/4943
676 https://github.com/STEllAR-GROUP/hpx/pull/4938
677 https://github.com/STEllAR-GROUP/hpx/pull/4858
678 https://github.com/STEllAR-GROUP/hpx/pull/4744
679 https://github.com/STEllAR-GROUP/hpx/pull/4665
680 https://github.com/STEllAR-GROUP/hpx/pull/4649
681 https://github.com/STEllAR-GROUP/hpx/pull/4642
682 https://github.com/STEllAR-GROUP/hpx/pull/4640
683 https://github.com/STEllAR-GROUP/hpx/pull/4622
2.10.4 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[^1] - Parallel sort fails to compile with C++20
- Issue #4950[^2] - Build with `HPX_WITH_PARCELPORT_ACTION_COUNTERS` fails
- Issue #4940[^3] - Codespell report for “HPX” (on fossies.org)

Closed pull requests

- PR #4982[^5] - Add page about citing HPX to documentation
- PR #4981[^6] - Adding the missing include
- PR #4974[^7] - Remove leftover format export hack
- PR #4972[^8] - Removing use of `get_temporary_buffer` and `return_temporary_buffer`
- PR #4963[^9] - Renaming files to avoid warnings from the vs build system
- PR #4951[^10] - Fixing build if `HPX_WITH_PARCELPORT_ACTION_COUNTERS=On`
- PR #4944[^12] - Fix typos reported by fossies codespell report
- PR #4941[^13] - Adding some explanation to README about how to cite HPX
- PR #4939[^14] - Small changes

[^1]: https://github.com/STEllAR-GROUP/hpx/issues/4971
[^2]: https://github.com/STEllAR-GROUP/hpx/issues/4950
[^3]: https://github.com/STEllAR-GROUP/hpx/issues/4940
[^4]: https://github.com/STEllAR-GROUP/hpx/issues/4937
[^5]: https://github.com/STEllAR-GROUP/hpx/pull/4982
[^6]: https://github.com/STEllAR-GROUP/hpx/pull/4981
[^7]: https://github.com/STEllAR-GROUP/hpx/pull/4974
[^8]: https://github.com/STEllAR-GROUP/hpx/pull/4972
[^9]: https://github.com/STEllAR-GROUP/hpx/pull/4963
[^10]: https://github.com/STEllAR-GROUP/hpx/pull/4951
[^11]: https://github.com/STEllAR-GROUP/hpx/pull/4946
[^12]: https://github.com/STEllAR-GROUP/hpx/pull/4944
[^13]: https://github.com/STEllAR-GROUP/hpx/pull/4941
[^14]: https://github.com/STErrAR-GROUP/hpx/pull/4939
2.10.5 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 #4514698 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.

698 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 #4918699 - Rename distributed_executors module
• Issue #4900700 - Adding JOSS status badge to README
• Issue #4897701 - Compiler warning, deprecated header used by HPX itself
• Issue #4886702 - A future bound to an action executing on a different locality doesn’t capture exception state
• Issue #4880703 - Undefined reference to main build error when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
• Issue #4877704 - hpx_main might not able to start hpx runtime properly
• Issue #4850705 - Issues creating templated component
• Issue #4829706 - Spack package & HPX_WITH_GENERIC_CONTEXT_COROUTINES
• Issue #4820707 - PAPI counters don’t work
• Issue #4818708 - HPX can’t be used with IO pool turned off
• Issue #4816709 - Build of HPX fails when find_package(Boost) is called before FetchContent_MakeAvailable(hpx)
• Issue #4813710 - HPX MPI Future failed
• Issue #4811711 - Remove HPX::hpx_no_wrap_main target before 1.5.0 release

699 https://github.com/STEllAR-GROUP/hpx/issues/4918
700 https://github.com/STEllAR-GROUP/hpx/issues/4900
701 https://github.com/STEllAR-GROUP/hpx/issues/4897
702 https://github.com/STEllAR-GROUP/hpx/issues/4886
703 https://github.com/STEllAR-GROUP/hpx/issues/4880
704 https://github.com/STEllAR-GROUP/hpx/issues/4877
705 https://github.com/STEllAR-GROUP/hpx/issues/4850
706 https://github.com/STEllAR-GROUP/hpx/issues/4829
707 https://github.com/STEllAR-GROUP/hpx/issues/4820
708 https://github.com/STEllAR-GROUP/hpx/issues/4818
709 https://github.com/STEllAR-GROUP/hpx/issues/4816
710 https://github.com/STEllAR-GROUP/hpx/issues/4813
711 https://github.com/STEllAR-GROUP/hpx/issues/4811
• Issue #4810\textsuperscript{712} - In hpx::for_each::invoke_projected the hpx::util::decay is misguided
• Issue #4787\textsuperscript{713} - transform\_inclusive\_scan gives incorrect results for non-commutative operator
• Issue #4786\textsuperscript{714} - transform\_inclusive\_scan tries to implicitly convert between types, instead of using the provided \texttt{conv} function
• Issue #4779\textsuperscript{715} - HPX build error with GCC 10.1
• Issue #4766\textsuperscript{716} - Move HPX.Compute functionality to experimental namespace
• Issue #4763\textsuperscript{717} - License file name
• Issue #4758\textsuperscript{718} - CMake profiling results
• Issue #4755\textsuperscript{719} - Building HPX with support for PAPI fails
• Issue #4754\textsuperscript{720} - CMake cache creation breaks when using HPX with mimalloc
• Issue #4752\textsuperscript{721} - HPX MPI Future build failed
• Issue #4746\textsuperscript{722} - Memory leak when using dataflow icw components
• Issue #4731\textsuperscript{723} - Bug in stencil example, calculation of locality IDs
• Issue #4723\textsuperscript{724} - Build fail with NETWORKING OFF
• Issue #4720\textsuperscript{725} - Add compatibility headers for modules that had their module headers implicitly generated in 1.4.1
• Issue #4719\textsuperscript{726} - Undeprecate some module headers
• Issue #4712\textsuperscript{727} - Rename HPX\_MPI\_WITH\_FUTURES option
• Issue #4709\textsuperscript{728} - Make deprecation warnings overridable in dependent projects
• Issue #4691\textsuperscript{729} - Suggestion to fix and enhance the thread\_mapper API
• Issue #4686\textsuperscript{730} - Fix tutorials examples
• Issue #4685\textsuperscript{731} - HPX distributed map fails to compile
• Issue #4680\textsuperscript{732} - Build error with HPX\_WITH\_DYNAMIC\_HPX\_MAIN=OFF
• Issue #4679\textsuperscript{733} - Build error for hpx w/ Apex on Summit

\textsuperscript{712} https://github.com/STEllAR-GROUP/hpx/issues/4810
\textsuperscript{713} https://github.com/STEllAR-GROUP/hpx/issues/4787
\textsuperscript{714} https://github.com/STEllAR-GROUP/hpx/issues/4786
\textsuperscript{715} https://github.com/STEllAR-GROUP/hpx/issues/4779
\textsuperscript{716} https://github.com/STEllAR-GROUP/hpx/issues/4766
\textsuperscript{717} https://github.com/STEllAR-GROUP/hpx/issues/4763
\textsuperscript{718} https://github.com/STEllAR-GROUP/hpx/issues/4758
\textsuperscript{719} https://github.com/STEllAR-GROUP/hpx/issues/4755
\textsuperscript{720} https://github.com/STEllAR-GROUP/hpx/issues/4754
\textsuperscript{721} https://github.com/STEllAR-GROUP/hpx/issues/4752
\textsuperscript{722} https://github.com/STEllAR-GROUP/hpx/issues/4746
\textsuperscript{723} https://github.com/STEllAR-GROUP/hpx/issues/4731
\textsuperscript{724} https://github.com/STEllAR-GROUP/hpx/issues/4723
\textsuperscript{725} https://github.com/STEllAR-GROUP/hpx/issues/4720
\textsuperscript{726} https://github.com/STEllAR-GROUP/hpx/issues/4719
\textsuperscript{727} https://github.com/STEllAR-GROUP/hpx/issues/4712
\textsuperscript{728} https://github.com/STEllAR-GROUP/hpx/issues/4709
\textsuperscript{729} https://github.com/STEllAR-GROUP/hpx/issues/4691
\textsuperscript{730} https://github.com/STEllAR-GROUP/hpx/issues/4686
\textsuperscript{731} https://github.com/STEllAR-GROUP/hpx/issues/4685
\textsuperscript{732} https://github.com/STEllAR-GROUP/hpx/issues/4680
\textsuperscript{733} https://github.com/STEllAR-GROUP/hpx/issues/4679
• Issue #4675 - Build error with HPX_WITH_NETWORKING=OFF
• Issue #4674 - Error running Quickstart tests on OS X
• Issue #4662 - MPI initialization broken when networking off
• Issue #4652 - How to fix distributed action annotation
• Issue #4650 - Thread descriptions are broken... again
• Issue #4648 - Thread stacksize not properly set
• Issue #4647 - Rename generated collective headers in modules
• Issue #4639 - Update deprecation warnings in compatibility headers to point to collective headers
• Issue #4628 - mpi parcelport totally broken
• Issue #4619 - Fully document hpx_wrap behaviour and targets
• Issue #4612 - Compilation issue with HPX 1.4.1 and 1.4.0
• Issue #4594 - Rename modules
• Issue #4578 - Default value for HPX_WITH_THREAD_BACKTRACE_DEPTH
• Issue #4572 - Thread manager should be given a runtime_configuration
• Issue #4571 - Add high-level documentation to new modules
• Issue #4569 - Annoying warning when compiling - pls suppress or fix it.
• Issue #4555 - HPX_HAVE_THREAD_BACKTRACE_ON_SUSPENSION compilation error
• Issue #4543 - Segfaults in Release builds using sleep_for
• Issue #4539 - Compilation Error when HPX_MPI_WITH_FUTURES=ON
• Issue #4537 - Linking issue with libhpx_initd.a
• Issue #4535 - API for checking if pool with a given name exists
• Issue #4523 - Build of PR #4311 (git tag 9955e8e) fails
• Issue #4519 - Documentation problem

734 https://github.com/STEllAR-GROUP/hpx/issues/4675
735 https://github.com/STEllAR-GROUP/hpx/issues/4674
736 https://github.com/STEllAR-GROUP/hpx/issues/4662
737 https://github.com/STEllAR-GROUP/hpx/issues/4652
738 https://github.com/STEllAR-GROUP/hpx/issues/4650
739 https://github.com/STEllAR-GROUP/hpx/issues/4648
740 https://github.com/STEllAR-GROUP/hpx/issues/4647
741 https://github.com/STEllAR-GROUP/hpx/issues/4639
742 https://github.com/STEllAR-GROUP/hpx/issues/4628
743 https://github.com/STEllAR-GROUP/hpx/issues/4619
744 https://github.com/STEllAR-GROUP/hpx/issues/4612
745 https://github.com/STEllAR-GROUP/hpx/issues/4594
746 https://github.com/STEllAR-GROUP/hpx/issues/4578
747 https://github.com/STEllAR-GROUP/hpx/issues/4572
748 https://github.com/STEllAR-GROUP/hpx/issues/4571
749 https://github.com/STEllAR-GROUP/hpx/issues/4569
750 https://github.com/STEllAR-GROUP/hpx/issues/4555
751 https://github.com/STEllAR-GROUP/hpx/issues/4543
752 https://github.com/STEllAR-GROUP/hpx/issues/4539
753 https://github.com/STEllAR-GROUP/hpx/issues/4537
754 https://github.com/STEllAR-GROUP/hpx/issues/4535
755 https://github.com/STEllAR-GROUP/hpx/issues/4523
756 https://github.com/STEllAR-GROUP/hpx/issues/4519
- Issue #4513: HPXConfig.cmake contains ill-formed paths when library paths use backslashes
- Issue #4507: User-polling introduced by MPI futures module should be more generally usable
- Issue #4506: Make sure force_linking.hpp is not included in main module header
- Issue #4501: Fix compilation of PAPI tests
- Issue #4497: Add modules CI checks
- Issue #4489: Polymorphic executor
- Issue #4476: Use CMake targets defined by FindBoost
- Issue #4473: Add vcpkg installation instructions
- Issue #4470: Adapt hpx::future to C++20 co_await
- Issue #4468: Compile error on Raspberry Pi 4
- Issue #4466: Compile error on Windows, current stable:
- Issue #4453: Installing HPX on fedora with dnf is not adding cmake files
- Issue #4448: New std::variant serialization broken
- Issue #4438: Add performance counter flag is monotonically increasing
- Issue #4436: Build problem: same code build and works with 1.4.0 but it doesn’t with 1.4.1
- Issue #4429: Function descriptions not supported in distributed
- Issue #4423: --hpx:ini=hpx.lock_detection=0 has no effect
- Issue #4422: Add performance counter metadata
- Issue #4419: Weird behavior for --hpx:print-counter-interval with large numbers
- Issue #4401: Create module repository
- Issue #4400: Command line options conflict related to performance counters
- Issue #4349: --hpx:use-process-mask option throw an exception on OS X
- Issue #4345: Move gh-pages branch out of hpx repo

757 https://github.com/STEllAR-GROUP/hpx/issues/4513
758 https://github.com/STEllAR-GROUP/hpx/issues/4507
759 https://github.com/STEllAR-GROUP/hpx/issues/4506
760 https://github.com/STEllAR-GROUP/hpx/issues/4501
761 https://github.com/STEllAR-GROUP/hpx/issues/4497
762 https://github.com/STEllAR-GROUP/hpx/issues/4489
763 https://github.com/STEllAR-GROUP/hpx/issues/4476
764 https://github.com/STEllAR-GROUP/hpx/issues/4473
765 https://github.com/STEllAR-GROUP/hpx/issues/4470
766 https://github.com/STEllAR-GROUP/hpx/issues/4468
767 https://github.com/STEllAR-GROUP/hpx/issues/4466
768 https://github.com/STEllAR-GROUP/hpx/issues/4453
769 https://github.com/STEllAR-GROUP/hpx/issues/4448
770 https://github.com/STEllAR-GROUP/hpx/issues/4438
771 https://github.com/STEllAR-GROUP/hpx/issues/4436
772 https://github.com/STEllAR-GROUP/hpx/issues/4429
773 https://github.com/STEllAR-GROUP/hpx/issues/4423
774 https://github.com/STEllAR-GROUP/hpx/issues/4422
775 https://github.com/STEllAR-GROUP/hpx/issues/4419
776 https://github.com/STEllAR-GROUP/hpx/issues/4401
777 https://github.com/STEllAR-GROUP/hpx/issues/4400
778 https://github.com/STEllAR-GROUP/hpx/issues/4349
779 https://github.com/STEllAR-GROUP/hpx/issues/4345
• Issue #4323 - Const-correctness error in assignment operator of compute::vector
• Issue #4318 - ASIO breaks with C++2a concepts
• Issue #4317 - Application runs even if --hpx:help is specified
• Issue #4063 - Document hpxcxx compiler wrapper
• Issue #3983 - Implement the C++20 Synchronization Library
• Issue #3698 - C++11 constexpr support is now required
• Issue #3623 - Modular HPX branch and an alternative project layout
• Issue #2836 - The worst-case time complexity of parallel::sort seems to be $O(N^2)$.

Closed pull requests

• PR #4936 - Minor documentation fixes part 2
• PR #4935 - Add copyright and license to joss paper file
• PR #4934 - Adding Semicolon in Documentation
• PR #4932 - Fixing compiler warnings
• PR #4931 - Small documentation formatting fixes
• PR #4930 - Documentation Distributed HPX applications localvv with local_vv
• PR #4929 - Add final version of the JOSS paper
• PR #4928 - Add HPX_NODISCARD to enable_user_polling structs
• PR #4926 - Rename distributed_executors module to executors_distributed
• PR #4925 - Making transform_reduce conforming to C++20
• PR #4923 - Don’t acquire lock if not needed
• PR #4921 - Update the release notes for the release candidate 3
• PR #4920 - Disable libcds release
• PR #4919 - Make cuda event pool dynamic instead of fixed size
- PR #4917 - Move chrono functionality to hpx::chrono namespace
- PR #4916 - HPX_HAVE_DEPRECATED_WARNINGS needs to be set even when disabled
- PR #4915 - Moving more action related files to actions modules
- PR #4914 - Add alias targets with namespaces used for exporting
- PR #4913 - Aggregate initialize CPOs
- PR #4910 - Explicitly specify hwloc root on Jenkins CSCS builds
- PR #4908 - Fix algorithms documentation
- PR #4907 - Remove HPX::hpx_no_wrap_main target
- PR #4906 - Fixing unused variable warning
- PR #4905 - Adding specializations for simple_for_loops
- PR #4904 - Update boost to 1.74.0 for the newest jenkins configs
- PR #4903 - Hide GITHUB_TOKEN environment variables from environment variable output
- PR #4902 - Cancel previous pull requests builds before starting a new one with Jenkins
- PR #4901 - Update public API list with updated algorithms
- PR #4899 - Suggested changes for HPX V1.5 release notes
- PR #4898 - Minor tweak to hpx::equal implementation
- PR #4896 - Making generate() and generate_n conforming to C++20
- PR #4895 - Update apex tag
- PR #4894 - Fix exception handling for tasks
- PR #4893 - Remove last use of std::result_of, removed in C++20
- PR #4892 - Adding replay_executor and replicate_executor
- PR #4890 - Restore old behaviour of not requiring linking to hpx_wrap when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4887 - Making sure remotely thrown (non-hpx) exceptions are properly marshaled back to invocation

PR #4917 [https://github.com/STEllAR-GROUP/hpx/pull/4917](https://github.com/STEllAR-GROUP/hpx/pull/4917)
PR #4916 [https://github.com/STEllAR-GROUP/hpx/pull/4916](https://github.com/STEllAR-GROUP/hpx/pull/4916)
PR #4915 [https://github.com/STEllAR-GROUP/hpx/pull/4915](https://github.com/STEllAR-GROUP/hpx/pull/4915)
PR #4914 [https://github.com/STEllAR-GROUP/hpx/pull/4914](https://github.com/STEllAR-GROUP/hpx/pull/4914)
PR #4913 [https://github.com/STEllAR-GROUP/hpx/pull/4913](https://github.com/STEllAR-GROUP/hpx/pull/4913)
PR #4910 [https://github.com/STEllAR-GROUP/hpx/pull/4910](https://github.com/STEllAR-GROUP/hpx/pull/4910)
PR #4908 [https://github.com/STEllAR-GROUP/hpx/pull/4908](https://github.com/STEllAR-GROUP/hpx/pull/4908)
PR #4906 [https://github.com/STEllAR-GROUP/hpx/pull/4906](https://github.com/STEllAR-GROUP/hpx/pull/4906)
PR #4905 [https://github.com/STEllAR-GROUP/hpx/pull/4905](https://github.com/STEllAR-GROUP/hpx/pull/4905)
PR #4904 [https://github.com/STEllAR-GROUP/hpx/pull/4904](https://github.com/STEllAR-GROUP/hpx/pull/4904)
PR #4903 [https://github.com/STEllAR-GROUP/hpx/pull/4903](https://github.com/STEllAR-GROUP/hpx/pull/4903)
PR #4902 [https://github.com/STEllAR-GROUP/hpx/pull/4902](https://github.com/STEllAR-GROUP/hpx/pull/4902)
PR #4901 [https://github.com/STEllAR-GROUP/hpx/pull/4901](https://github.com/STEllAR-GROUP/hpx/pull/4901)
PR #4899 [https://github.com/STEllAR-GROUP/hpx/pull/4899](https://github.com/STEllAR-GROUP/hpx/pull/4899)
PR #4898 [https://github.com/STEllAR-GROUP/hpx/pull/4898](https://github.com/STEllAR-GROUP/hpx/pull/4898)
PR #4896 [https://github.com/STEllAR-GROUP/hpx/pull/4896](https://github.com/STEllAR-GROUP/hpx/pull/4896)
PR #4895 [https://github.com/STEllAR-GROUP/hpx/pull/4895](https://github.com/STEllAR-GROUP/hpx/pull/4895)
PR #4894 [https://github.com/STEllAR-GROUP/hpx/pull/4894](https://github.com/STEllAR-GROUP/hpx/pull/4894)
PR #4893 [https://github.com/STEllAR-GROUP/hpx/pull/4893](https://github.com/STEllAR-GROUP/hpx/pull/4893)
PR #4892 [https://github.com/STEllAR-GROUP/hpx/pull/4892](https://github.com/STEllAR-GROUP/hpx/pull/4892)
PR #4889 [https://github.com/STEllAR-GROUP/hpx/pull/4889](https://github.com/STEllAR-GROUP/hpx/pull/4889)
PR #4887 [https://github.com/STEllAR-GROUP/hpx/pull/4887](https://github.com/STEllAR-GROUP/hpx/pull/4887)
site

- PR #4885 - Adapting hpx::find and friends to C++20
- PR #4884 - Adapting mismatch to C++20
- PR #4883 - Adapting hpx::equal to be conforming to C++20
- PR #4882 - Fixing exception handling for hpx::copy and adding missing tests
- PR #4881 - Adds different runtime exception when registering thread with the HPX runtime
- PR #4876 - Adding example demonstrating how to disable thread stealing during the execution of parallel algorithms
- PR #4874 - Adding non-policy tests to all_of, any_of, and none_of
- PR #4873 - Set CUDA compute capability on rostam Jenkins builds
- PR #4872 - Force partitioned vector scan tests to run serially
- PR #4870 - Making move conforming with C++20
- PR #4869 - Making destroy and destroy_n conforming to C++20
- PR #4868 - Fix miscellaneous header problems
- PR #4867 - Add CPOs for for_each
- PR #4865 - Adapting count and count_if to be conforming to C++20
- PR #4864 - Release notes 1.5.0
- PR #4863 - adding libeds-hpx tag to prepare for hpx1.5 release
- PR #4862 - Adding version specific deprecation options
- PR #4861 - Limiting executor improvements
- PR #4860 - Making fill and fill_n compatible with C++20
- PR #4859 - Adapting all_of, any_of, and none_of to C++20
- PR #4858 - Improve libCDS integration
- PR #4856 - Correct typos in the documentation of the hpx performance counters

https://github.com/STEllAR-GROUP/hpx/pull/4885
https://github.com/STEllAR-GROUP/hpx/pull/4884
https://github.com/STEllAR-GROUP/hpx/pull/4883
https://github.com/STEllAR-GROUP/hpx/pull/4882
https://github.com/STEllAR-GROUP/hpx/pull/4881
https://github.com/STEllAR-GROUP/hpx/pull/4876
https://github.com/STEllAR-GROUP/hpx/pull/4874
https://github.com/STEllAR-GROUP/hpx/pull/4873
https://github.com/STEllAR-GROUP/hpx/pull/4872
https://github.com/STEllAR-GROUP/hpx/pull/4870
https://github.com/STEllAR-GROUP/hpx/pull/4869
https://github.com/STEllAR-GROUP/hpx/pull/4868
https://github.com/STEllAR-GROUP/hpx/pull/4867
https://github.com/STEllAR-GROUP/hpx/pull/4865
https://github.com/STEllAR-GROUP/hpx/pull/4864
https://github.com/STEllAR-GROUP/hpx/pull/4863
https://github.com/STEllAR-GROUP/hpx/pull/4862
https://github.com/STEllAR-GROUP/hpx/pull/4861
https://github.com/STEllAR-GROUP/hpx/pull/4860
https://github.com/STEllAR-GROUP/hpx/pull/4859
https://github.com/STEllAR-GROUP/hpx/pull/4857
https://github.com/STEllAR-GROUP/hpx/pull/4856
• PR #4854 - Removing obsolete code
• PR #4853 - Adding test that derives component from two other components
• PR #4852 - Fix mpi_ring test in distributed mode by ensuring all ranks run hpx_main
• PR #4851 - Converting resiliency APIs to tag_invoke based CPOs
• PR #4849 - Enable use of future_overhead test when DISTRIBUTED_RUNTIME is OFF
• PR #4847 - Fixing 'error prone' constructs as reported by Codacy
• PR #4846 - Disable Boost.Asio concepts support
• PR #4845 - Fix PAPI counters
• PR #4843 - Remove dependency on various Boost headers
• PR #4841 - Rearrange public API headers
• PR #4840 - Fixing TSS problems during thread termination
• PR #4839 - Fix async_cuda build problems when distributed runtime is disabled
• PR #4837 - Restore compatibility for old (now deprecated) copy algorithms
• PR #4836 - Adding CPOs for hpx::reduce
• PR #4835 - Remove using util::result_of from namespace hpx
• PR #4834 - Fixing the calculation of the number of idle cores and the corresponding idle masks
• PR #4833 - Allow thread function destructors to yield
• PR #4832 - Fixing assertion in split_gids and memory leaks in 1d_stencil_7
• PR #4831 - Making sure MPI_CXX_COMPILE_FLAGS is interpreted as a sequence of options
• PR #4830 - Update documentation on using HPX::wrap_main
• PR #4827 - Update clang-newest configuration to use clang 10
• PR #4826 - Add Jenkins configuration for rostam
• PR #4825 - Move all CUDA functionality to hpx::cuda::experimental namespace

847 https://github.com/STEllAR-GROUP/hpx/pull/4854
848 https://github.com/STEllAR-GROUP/hpx/pull/4853
849 https://github.com/STEllAR-GROUP/hpx/pull/4852
850 https://github.com/STEllAR-GROUP/hpx/pull/4851
851 https://github.com/STEllAR-GROUP/hpx/pull/4849
852 https://github.com/STEllAR-GROUP/hpx/pull/4847
853 https://github.com/STEllAR-GROUP/hpx/pull/4846
854 https://github.com/STEllAR-GROUP/hpx/pull/4845
855 https://github.com/STEllAR-GROUP/hpx/pull/4843
856 https://github.com/STEllAR-GROUP/hpx/pull/4841
857 https://github.com/STEllAR-GROUP/hpx/pull/4840
858 https://github.com/STEllAR-GROUP/hpx/pull/4839
859 https://github.com/STEllAR-GROUP/hpx/pull/4837
860 https://github.com/STEllAR-GROUP/hpx/pull/4836
861 https://github.com/STEllAR-GROUP/hpx/pull/4835
862 https://github.com/STEllAR-GROUP/hpx/pull/4834
863 https://github.com/STEllAR-GROUP/hpx/pull/4833
864 https://github.com/STEllAR-GROUP/hpx/pull/4832
865 https://github.com/STEllAR-GROUP/hpx/pull/4831
866 https://github.com/STEllAR-GROUP/hpx/pull/4830
867 https://github.com/STEllAR-GROUP/hpx/pull/4827
868 https://github.com/STEllAR-GROUP/hpx/pull/4826
869 https://github.com/STEllAR-GROUP/hpx/pull/4825

3.10. Releases
• PR #4824 - Add support for building master/release branches to Jenkins configuration
• PR #4821 - Implement customization point for hpx::copy and hpx::ranges::copy
• PR #4819 - Allow finding Boost components before finding HPX
• PR #4817 - Adding range version of stable sort
• PR #4815 - Fix a wrong #ifdef for IO/TIMER pools causing build errors
• PR #4814 - Replace hpx::function_nonser with std::function in error module
• PR #4809 - Foreach adapt
• PR #4808 - Make internal algorithms functions const
• PR #4807 - Add Jenkins configuration for running on Piz Daint
• PR #4806 - Update documentation links to new domain name
• PR #4805 - Applying changes that resolve time complexity issues in sort
• PR #4803 - Adding implementation of stable_sort
• PR #4802 - Fix datapar header paths
• PR #4801 - Replace boost::shared_array<T> with std::shared_ptr<T[]> if supported
• PR #4799 - Fixing #include paths in compatibility headers
• PR #4798 - Include the main module header (fixes partially #4488)
• PR #4797 - Change cmake targets
• PR #4794 - Removing 128bit integer emulation
• PR #4793 - Make sure global variable is handled properly
• PR #4792 - Replace enable_if with HPX_CONCEPT_REQUIRES_ and add is_sentinel_for constraint
• PR #4790 - Move deprecation warnings from base template to template specializations for result_of etc. structs
• PR #4789 - Fix hangs during assertion handling and distributed runtime construction
• PR #4788 - Fixing inclusive transform scan algorithm to properly handle initial value

870 https://github.com/STEllAR-GROUP/hpx/pull/4824
871 https://github.com/STEllAR-GROUP/hpx/pull/4821
872 https://github.com/STEllAR-GROUP/hpx/pull/4819
873 https://github.com/STEllAR-GROUP/hpx/pull/4817
874 https://github.com/STEllAR-GROUP/hpx/pull/4815
875 https://github.com/STEllAR-GROUP/hpx/pull/4814
876 https://github.com/STEllAR-GROUP/hpx/pull/4809
877 https://github.com/STEllAR-GROUP/hpx/pull/4808
878 https://github.com/STEllAR-GROUP/hpx/pull/4807
879 https://github.com/STEllAR-GROUP/hpx/pull/4806
880 https://github.com/STEllAR-GROUP/hpx/pull/4805
881 https://github.com/STEllAR-GROUP/hpx/pull/4803
882 https://github.com/STEllAR-GROUP/hpx/pull/4802
883 https://github.com/STEllAR-GROUP/hpx/pull/4801
884 https://github.com/STEllAR-GROUP/hpx/pull/4799
885 https://github.com/STEllAR-GROUP/hpx/pull/4798
886 https://github.com/STEllAR-GROUP/hpx/pull/4797
887 https://github.com/STEllAR-GROUP/hpx/pull/4794
888 https://github.com/STEllAR-GROUP/hpx/pull/4793
889 https://github.com/STEllAR-GROUP/hpx/pull/4792
890 https://github.com/STEllAR-GROUP/hpx/pull/4790
891 https://github.com/STEllAR-GROUP/hpx/pull/4789
892 https://github.com/STEllAR-GROUP/hpx/pull/4788

Chapter 2. What’s so special about HPX?
• PR #4785[^93] - Fixing barrier test
• PR #4784[^94] - Fixing deleter argument bindings in serialize_buffer
• PR #4783[^95] - Add coveralls badge
• PR #4782[^96] - Make header tests parallel again
• PR #4780[^97] - Remove outdated comment about hpx::stop in documentation
• PR #4776[^98] - debug print improvements
• PR #4775[^99] - Checkpoint cleanup
• PR #4771[^100] - Fix compilation with HPX_WITH_NETWORKING=OFF
• PR #4767[^101] - Remove all force linking leftovers
• PR #4765[^102] - Fix 1d stencil index calculation
• PR #4764[^103] - Force some tests to run serially
• PR #4762[^104] - Update pointees in compatibility headers
• PR #4761[^105] - Fix running and building of execution module tests on CircleCI
• PR #4760[^106] - Storing hpx_options in global property to speed up summary report
• PR #4759[^107] - Reduce memory requirements for our main shared state
• PR #4757[^108] - Fix mimalloc linking on Windows
• PR #4756[^109] - Fix compilation issues
• PR #4753[^110] - Re-adding API functions that were lost during merges
• PR #4751[^111] - Revert “Create coverage reports and upload them to codecov.io”
• PR #4750[^112] - Fixing possible race condition during termination detection
• PR #4749[^113] - Deprecate result_of and friends
• PR #4748[^114] - Create coverage reports and upload them to codecov.io
• PR #4747[^115] - Changing #include for MPI parcelport

[^93]: https://github.com/STEllAR-GROUP/hpx/pull/4785
[^94]: https://github.com/STEllAR-GROUP/hpx/pull/4784
[^95]: https://github.com/STEllAR-GROUP/hpx/pull/4783
[^96]: https://github.com/STEllAR-GROUP/hpx/pull/4782
[^97]: https://github.com/STEllAR-GROUP/hpx/pull/4776
[^98]: https://github.com/STEllAR-GROUP/hpx/pull/4775
[^99]: https://github.com/STEllAR-GROUP/hpx/pull/4771
[^100]: https://github.com/STEllAR-GROUP/hpx/pull/4767
[^101]: https://github.com/STEllAR-GROUP/hpx/pull/4765
[^102]: https://github.com/STEllAR-GROUP/hpx/pull/4764
[^103]: https://github.com/STEllAR-GROUP/hpx/pull/4762
[^104]: https://github.com/STEllAR-GROUP/hpx/pull/4759
[^105]: https://github.com/STEllAR-GROUP/hpx/pull/4757
[^106]: https://github.com/STEllAR-GROUP/hpx/pull/4756
[^107]: https://github.com/STEllAR-GROUP/hpx/pull/4753
[^108]: https://github.com/STEllAR-GROUP/hpx/pull/4751
[^109]: https://github.com/STEllAR-GROUP/hpx/pull/4750
[^110]: https://github.com/STEllAR-GROUP/hpx/pull/4749
[^111]: https://github.com/STEllAR-GROUP/hpx/pull/4748
[^112]: https://github.com/STEllAR-GROUP/hpx/pull/4747

2.10. Releases
• PR #4745 - Add is_sentinel_for trait implementation and test
• PR #4743 - Fix init_globally example after runtime mode changes
• PR #4742 - Update SUPPORT.md
• PR #4741 - Fixing a warning generated for unity builds with msvc
• PR #4740 - Rename local_lcos and basic_execution modules
• PR #4739 - Undeprecate a couple of hpx/modulename.hpp headers
• PR #4738 - Conditionally test schedulers in thread_stacksize_current test
• PR #4734 - Fixing a bunch of codacy warnings
• PR #4733 - Add experimental unity build option to CMake configuration
• PR #4732 - Fixing compilation problems with unordered map
• PR #4729 - Fix APEX build
• PR #4727 - Fix missing runtime includes for distributed runtime
• PR #4726 - Add more API headers
• PR #4725 - Add more compatibility headers for deprecated module headers
• PR #4724 - Fix 4723
• PR #4721 - Attempt to fixing migration tests
• PR #4719 - Make the compatibility headers macro conditional
• PR #4716 - Add hpx/runtime.hpp and hpx/distributed/runtime.hpp API headers
• PR #4714 - Add hpx/future.hpp header
• PR #4713 - Remove hpx/runtime/threads_fwd.hpp and hpx/util_fwd.hpp
• PR #4711 - Make module deprecation warnings overridable
• PR #4710 - Add compatibility headers and other fixes after module header renaming
• PR #4708 - Add termination handler for parallel algorithms

916 https://github.com/STEllAR-GROUP/hpx/pull/4745
917 https://github.com/STEllAR-GROUP/hpx/pull/4743
918 https://github.com/STEllAR-GROUP/hpx/pull/4742
919 https://github.com/STEllAR-GROUP/hpx/pull/4741
920 https://github.com/STEllAR-GROUP/hpx/pull/4740
921 https://github.com/STEllAR-GROUP/hpx/pull/4739
922 https://github.com/STEllAR-GROUP/hpx/pull/4738
923 https://github.com/STEllAR-GROUP/hpx/pull/4734
924 https://github.com/STEllAR-GROUP/hpx/pull/4733
925 https://github.com/STEllAR-GROUP/hpx/pull/4730
926 https://github.com/STEllAR-GROUP/hpx/pull/4729
927 https://github.com/STEllAR-GROUP/hpx/pull/4727
928 https://github.com/STEllAR-GROUP/hpx/pull/4726
929 https://github.com/STEllAR-GROUP/hpx/pull/4725
930 https://github.com/STEllAR-GROUP/hpx/pull/4724
931 https://github.com/STEllAR-GROUP/hpx/pull/4721
932 https://github.com/STEllAR-GROUP/hpx/pull/4717
933 https://github.com/STEllAR-GROUP/hpx/pull/4716
934 https://github.com/STEllAR-GROUP/hpx/pull/4714
935 https://github.com/STEllAR-GROUP/hpx/pull/4713
936 https://github.com/STEllAR-GROUP/hpx/pull/4711
937 https://github.com/STEllAR-GROUP/hpx/pull/4710
938 https://github.com/STEllAR-GROUP/hpx/pull/4708
- PR #4707 - Use hpx::function_nonser instead of std::function internally
- PR #4706 - Move header file to module
- PR #4705 - Fix incorrect behaviour of cmake-format check
- PR #4704 - Fix resource tests
- PR #4701 - Fix missing includes for future::then specializations
- PR #4700 - Removing obsolete memory component
- PR #4699 - Add short descriptions to modules missing documentation
- PR #4696 - Rename generated modules headers
- PR #4693 - Overhauling thread_mapper for public consumption
- PR #4688 - Fix thread stack size handling
- PR #4687 - Adding all_gather and fixing all_to_all
- PR #4684 - Miscellaneous compilation fixes
- PR #4683 - Fix HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4682 - Fix compilation of pack_traversal_rebind_container.hpp
- PR #4681 - Add missing hpx execution.hpp includes for future::then
- PR #4678 - Typeless communicator
- PR #4677 - Forcing registry option to be accepted without checks.
- PR #4676 - Adding scatter_to/scatter_from collective operations
- PR #4673 - Fix PAPI counters compilation
- PR #4671 - Deprecate hpx::promise alias to hpx::lcos::promise
- PR #4670 - Explicitly instantiate get_exception
- PR #4667 - Add stopValue in Sentinel struct instead of Iterator
- PR #4666 - Add release build on Windows to GitHub actions

939 https://github.com/STEllAR-GROUP/hpx/pull/4707
940 https://github.com/STEllAR-GROUP/hpx/pull/4706
941 https://github.com/STEllAR-GROUP/hpx/pull/4705
942 https://github.com/STEllAR-GROUP/hpx/pull/4704
943 https://github.com/STEllAR-GROUP/hpx/pull/4700
944 https://github.com/STEllAR-GROUP/hpx/pull/4699
946 https://github.com/STEllAR-GROUP/hpx/pull/4696
947 https://github.com/STEllAR-GROUP/hpx/pull/4693
948 https://github.com/STEllAR-GROUP/hpx/pull/4688
949 https://github.com/STEllAR-GROUP/hpx/pull/4687
950 https://github.com/STEllAR-GROUP/hpx/pull/4684
951 https://github.com/STEllAR-GROUP/hpx/pull/4683
952 https://github.com/STEllAR-GROUP/hpx/pull/4682
953 https://github.com/STEllAR-GROUP/hpx/pull/4681
954 https://github.com/STEllAR-GROUP/hpx/pull/4678
955 https://github.com/STEllAR-GROUP/hpx/pull/4677
956 https://github.com/STEllAR-GROUP/hpx/pull/4676
957 https://github.com/STEllAR-GROUP/hpx/pull/4673
958 https://github.com/STEllAR-GROUP/hpx/pull/4671
959 https://github.com/STEllAR-GROUP/hpx/pull/4670
960 https://github.com/STEllAR-GROUP/hpx/pull/4667
961 https://github.com/STEllAR-GROUP/hpx/pull/4666

2.10. Releases
• PR #4664 - Creating itt_notify module.
• PR #4663 - Mpi fixes
• PR #4659 - Making sure declarations match definitions in register_locks implementation
• PR #4655 - Fixing task annotations for actions
• PR #4653 - Making sure APEX is linked into every application, if needed
• PR #4651 - Update get_function_annotation.hpp
• PR #4646 - Runtime type
• PR #4645 - Add a few more API headers
• PR #4644 - Fixing support for mpirun (and similar)
• PR #4638 - Fixing the fix for get_idle_core_count() API
• PR #4637 - Remove HPX_API_EXPORT missed in previous cleanup
• PR #4636 - Adding C++20 barrier
• PR #4635 - Adding C++20 latch API
• PR #4634 - Adding C++20 counting semaphore API
• PR #4633 - Unify execution parameters customization points
• PR #4632 - Adding missing bulk_sync_execute wrapper to example executor
• PR #4631 - Updates to documentation; grammar edits.
• PR #4630 - Updates to documentation; moved hyperlink
• PR #4624 - Export set_self_ptr in thread_data.hpp instead of with forward declarations where used
• PR #4623 - Clean up export macros
• PR #4621 - Trigger an error for older boost versions on power architectures
• PR #4617 - Ignore user-set compatibility header options if the module does not have compatibility headers
• PR #4616 - Fix cmake-format warning

962 https://github.com/STEllAR-GROUP/hpx/pull/4664
963 https://github.com/STEllAR-GROUP/hpx/pull/4663
964 https://github.com/STEllAR-GROUP/hpx/pull/4659
965 https://github.com/STEllAR-GROUP/hpx/pull/4655
966 https://github.com/STEllAR-GROUP/hpx/pull/4653
967 https://github.com/STEllAR-GROUP/hpx/pull/4651
968 https://github.com/STEllAR-GROUP/hpx/pull/4646
969 https://github.com/STEllAR-GROUP/hpx/pull/4645
970 https://github.com/STEllAR-GROUP/hpx/pull/4644
971 https://github.com/STEllAR-GROUP/hpx/pull/4643
972 https://github.com/STEllAR-GROUP/hpx/pull/4638
973 https://github.com/STEllAR-GROUP/hpx/pull/4636
974 https://github.com/STEllAR-GROUP/hpx/pull/4635
975 https://github.com/STEllAR-GROUP/hpx/pull/4634
976 https://github.com/STEllAR-GROUP/hpx/pull/4633
977 https://github.com/STEllAR-GROUP/hpx/pull/4632
978 https://github.com/STEllAR-GROUP/hpx/pull/4631
979 https://github.com/STEllAR-GROUP/hpx/pull/4630
980 https://github.com/STEllAR-GROUP/hpx/pull/4624
981 https://github.com/STEllAR-GROUP/hpx/pull/4623
982 https://github.com/STEllAR-GROUP/hpx/pull/4621
983 https://github.com/STEllAR-GROUP/hpx/pull/4617
984 https://github.com/STEllAR-GROUP/hpx/pull/4616
• PR #4615\(^{985}\) - Add handler for serializing custom exceptions
• PR #4614\(^{986}\) - Fix error message when HPX_IGNORE_CMAKE_BUILD_TYPE_COMPATIBILITY=OFF
• PR #4613\(^{987}\) - Make partitioner constructor private
• PR #4611\(^{988}\) - Making auto_chunk_size execute the given function using the given executor
• PR #4610\(^{989}\) - Making sure the thread-local lock registration data is moving to the core the suspended HPX thread is resumed on
• PR #4609\(^{990}\) - Adding an API function that exposes the number of idle cores
• PR #4608\(^{991}\) - Fixing moodycamel namespace
• PR #4607\(^{992}\) - Moving winsocket initialization to core library
• PR #4606\(^{993}\) - Local runtime module etc.
• PR #4604\(^{994}\) - Add config_registry module
• PR #4603\(^{995}\) - Deal with distributed modules in their respective CMakeLists.txt
• PR #4602\(^{996}\) - Small module fixes
• PR #4598\(^{997}\) - Making sure current_executor and service_executor functions are linked into the core library
• PR #4597\(^{998}\) - Adding broadcast_to/broadcast_from to collectives module
• PR #4596\(^{999}\) - Fix performance regression in block_executor
• PR #4595\(^{1000}\) - Making sure main.cpp is built as a library if HPX_WITH_DYNAMIC_MAIN=OFF
• PR #4592\(^{1001}\) - Futures module
• PR #4591\(^{1002}\) - Adapting co_await support for C++20
• PR #4590\(^{1003}\) - Adding missing exception test for for_loop()
• PR #4587\(^{1004}\) - Move traits headers to hpx/modulename/traits directory
• PR #4586\(^{1005}\) - Remove Travis CI config
• PR #4585\(^{1006}\) - Update macOS test blacklist
• PR #4584\(^{1007}\) - Attempting to fix missing symbols in stack trace

\(^{985}\) https://github.com/STEllAR-GROUP/hpx/pull/4615
\(^{986}\) https://github.com/STEllAR-GROUP/hpx/pull/4614
\(^{987}\) https://github.com/STEllAR-GROUP/hpx/pull/4613
\(^{988}\) https://github.com/STEllAR-GROUP/hpx/pull/4611
\(^{989}\) https://github.com/STEllAR-GROUP/hpx/pull/4610
\(^{990}\) https://github.com/STEllAR-GROUP/hpx/pull/4609
\(^{991}\) https://github.com/STEllAR-GROUP/hpx/pull/4608
\(^{992}\) https://github.com/STEllAR-GROUP/hpx/pull/4607
\(^{993}\) https://github.com/STEllAR-GROUP/hpx/pull/4606
\(^{994}\) https://github.com/STEllAR-GROUP/hpx/pull/4604
\(^{995}\) https://github.com/STEllAR-GROUP/hpx/pull/4603
\(^{996}\) https://github.com/STEllAR-GROUP/hpx/pull/4602
\(^{997}\) https://github.com/STEllAR-GROUP/hpx/pull/4598
\(^{998}\) https://github.com/STEllAR-GROUP/hpx/pull/4597
\(^{999}\) https://github.com/STEllAR-GROUP/hpx/pull/4596
\(^{1000}\) https://github.com/STEllAR-GROUP/hpx/pull/4595
\(^{1001}\) https://github.com/STEllAR-GROUP/hpx/pull/4592
\(^{1002}\) https://github.com/STEllAR-GROUP/hpx/pull/4591
\(^{1003}\) https://github.com/STEllAR-GROUP/hpx/pull/4590
\(^{1004}\) https://github.com/STEllAR-GROUP/hpx/pull/4587
\(^{1005}\) https://github.com/STEllAR-GROUP/hpx/pull/4586
\(^{1006}\) https://github.com/STEllAR-GROUP/hpx/pull/4585
\(^{1007}\) https://github.com/STEllAR-GROUP/hpx/pull/4584
Chapter 2. What’s so special about HPX?
• PR #4538 - Move channel documentation examples to examples directory
• PR #4536 - Add generic allocator for execution policies
• PR #4534 - Enable compatibility headers for thread_executors module
• PR #4532 - Fixing broken url in README.rst
• PR #4531 - Update scripts
• PR #4530 - Make sure module API docs show up in correct order
• PR #4529 - Adding missing template code to module creation script
• PR #4528 - Make sure version module uses HPX’s binary dir, not the parent’s
• PR #4527 - Creating actions_base and actions module
• PR #4526 - Shared state for cv
• PR #4525 - Changing sub-name sequencing for experimental namespace
• PR #4524 - Add API guarantee notes to API reference documentation
• PR #4522 - Enable and fix deprecation warnings in execution module
• PR #4521 - Moves more miscellaneous files to modules
• PR #4520 - Skip execution customization points when executor is known
• PR #4518 - Module distributed lcos
• PR #4516 - Fix various builds
• PR #4515 - Replace backslashes by slashes in windows paths
• PR #4514 - Adding polymorphic_executor
• PR #4512 - Adding C++20 jthread and stop_token
• PR #4510 - Attempt to fix APEX linking in external packages again
• PR #4508 - Only test pull requests (not all branches) with GitHub actions
• PR #4505 - Fix duplicate linking in tests (ODR violations)

https://github.com/STEllAR-GROUP/hpx/pull/4538
https://github.com/STEllAR-GROUP/hpx/pull/4536
https://github.com/STEllAR-GROUP/hpx/pull/4534
https://github.com/STEllAR-GROUP/hpx/pull/4532
https://github.com/STEllAR-GROUP/hpx/pull/4531
https://github.com/STEllAR-GROUP/hpx/pull/4530
https://github.com/STEllAR-GROUP/hpx/pull/4529
https://github.com/STEllAR-GROUP/hpx/pull/4528
https://github.com/STEllAR-GROUP/hpx/pull/4527
https://github.com/STEllAR-GROUP/hpx/pull/4526
https://github.com/STEllAR-GROUP/hpx/pull/4525
https://github.com/STEllAR-GROUP/hpx/pull/4524
https://github.com/STEllAR-GROUP/hpx/pull/4523
https://github.com/STEllAR-GROUP/hpx/pull/4522
https://github.com/STEllAR-GROUP/hpx/pull/4521
https://github.com/STEllAR-GROUP/hpx/pull/4520
https://github.com/STEllAR-GROUP/hpx/pull/4518
https://github.com/STEllAR-GROUP/hpx/pull/4516
https://github.com/STEllAR-GROUP/hpx/pull/4515
https://github.com/STEllAR-GROUP/hpx/pull/4514
https://github.com/STEllAR-GROUP/hpx/pull/4513
https://github.com/STEllAR-GROUP/hpx/pull/4512
https://github.com/STEllAR-GROUP/hpx/pull/4510
https://github.com/STEllAR-GROUP/hpx/pull/4508
https://github.com/STEllAR-GROUP/hpx/pull/4505
• PR #4504 - Fix C++ standard handling
• PR #4503 - Add CMakelists file check
• PR #4500 - Fix .clang-format version requirement comment
• PR #4499 - Attempting to fix hpx_init linking on macOS
• PR #4498 - Fix compatibility of pool_executor
• PR #4496 - Removing superfluous SPDX tags
• PR #4494 - Module executors
• PR #4493 - Pack traversal module
• PR #4492 - Update copyright year in documentation
• PR #4491 - Add missing current_executor header
• PR #4490 - Update GitHub actions configs
• PR #4487 - Properly dispatch exceptions thrown from hpx_main to be rethrown from hpx::init/hpx::stop
• PR #4486 - Fixing an initialization order problem
• PR #4485 - Move miscellaneous files to their rightful modules
• PR #4483 - Clean up imported CMake target naming
• PR #4481 - Add vcpkg installation instructions
• PR #4479 - Add hints to allow to specify MIMALLOC_ROOT
• PR #4478 - Async modules
• PR #4475 - Fix rp init changes
• PR #4474 - Use #pragma once in headers
• PR #4472 - Add more descriptive error message when using x86 coroutines on non-x86 platforms
• PR #4467 - Add malloc find cmake script
• PR #4465 - Add thread_executors module
- Include module
- Merge hpx_init and hpx_wrap into one static library
- Making thread_data test more realistic
- Suppress MPI warnings in version.cpp
- Make sure pkgconfig applications link with hpx_init
- Added example demonstrating how to create and use a wrapping executor
- Fixing execution of thread exit functions
- Move backtrace files to debugging module
- Move deadlock_detection and maintain_queue_wait_times source files into schedulers module
- Fixing compilation with std::filesystem enabled
- Fix build system to actually build variant test
- This fixes an obsolete #include
- Resume tasks where they were suspended
- Minor CUDA fixes
- Add missing tests to CircleCI config
- Adding a tag to all auto-generated files allowing for tools to visually distinguish those
- Adding performance counter type information
- Fixing MSVC build
- Link HPX::plugin and component privately in hpx_setup_target
- Adding a test that verifies the problem can be solved using a trait specialization
- Clean up Boost dependencies and copy string algorithms to new module
- Fixing compilation issues (!) if MPI parcelport is enabled
- Ignore warnings about name mangling changing

1077 https://github.com/STEllAR-GROUP/hpx/pull/4464
1078 https://github.com/STEllAR-GROUP/hpx/pull/4462
1079 https://github.com/STEllAR-GROUP/hpx/pull/4461
1080 https://github.com/STEllAR-GROUP/hpx/pull/4460
1081 https://github.com/STEllAR-GROUP/hpx/pull/4459
1082 https://github.com/STEllAR-GROUP/hpx/pull/4458
1083 https://github.com/STEllAR-GROUP/hpx/pull/4457
1084 https://github.com/STEllAR-GROUP/hpx/pull/4456
1085 https://github.com/STEllAR-GROUP/hpx/pull/4455
1086 https://github.com/STEllAR-GROUP/hpx/pull/4450
1087 https://github.com/STEllAR-GROUP/hpx/pull/4449
1088 https://github.com/STEllAR-GROUP/hpx/pull/4447
1089 https://github.com/STEllAR-GROUP/hpx/pull/4444
1090 https://github.com/STEllAR-GROUP/hpx/pull/4443
1091 https://github.com/STEllAR-GROUP/hpx/pull/4442
1092 https://github.com/STEllAR-GROUP/hpx/pull/4441
1093 https://github.com/STEllAR-GROUP/hpx/pull/4440
1094 https://github.com/STEllAR-GROUP/hpx/pull/4439
1095 https://github.com/STEllAR-GROUP/hpx/pull/4438
1096 https://github.com/STEllAR-GROUP/hpx/pull/4437
1097 https://github.com/STEllAR-GROUP/hpx/pull/4436
1098 https://github.com/STEllAR-GROUP/hpx/pull/4435
1099 https://github.com/STEllAR-GROUP/hpx/pull/4434
• PR #4430 - Add performance_counters module
• PR #4428 - Don’t add compatibility headers to module API reference
• PR #4426 - Add currently failing tests on GitHub actions to blacklist
• PR #4425 - Clean up and correct minimum required versions
• PR #4424 - Making sure hpx.lock_detection=0 works as advertised
• PR #4421 - Making sure interval time stops underlying timer thread on termination
• PR #4417 - Adding serialization support for std::variant (if available) and std::tuple
• PR #4415 - Partially reverting changes applied by PR 4373
• PR #4414 - Added documentation for the compiler-wrapper script hpxcxx.in in creating_hpx_projects.rst
• PR #4413 - Merging from V1.4.1 release
• PR #4412 - Making sure to issue a warning if a file specified using --hpx:options-file is not found
• PR #4411 - Make test specific to HPX_WITH_SHARED_PRIORITY_SCHEDULER
• PR #4407 - Adding minimal MPI executor
• PR #4405 - Fix cross pool injection test, use default scheduler as fallback
• PR #4404 - Fix a race condition and clean-up usage of scheduler mode
• PR #4399 - Add more threading modules
• PR #4398 - Add CODEOWNERS file
• PR #4395 - Adding a parameter to auto_chunk_size allowing to control the amount of iterations to measure
• PR #4393 - Use appropriate cache-line size defaults for different platforms
• PR #4391 - Fixing use of allocator for C++20
• PR #4390 - Making --hpx:help behavior consistent
• PR #4388 - Change the resource partitioner initialization
• PR #4387 - Fix roll_release.sh

1100 https://github.com/STEllAR-GROUP/hpx/pull/4430
1101 https://github.com/STEllAR-GROUP/hpx/pull/4428
1102 https://github.com/STEllAR-GROUP/hpx/pull/4426
1103 https://github.com/STEllAR-GROUP/hpx/pull/4425
1104 https://github.com/STEllAR-GROUP/hpx/pull/4424
1105 https://github.com/STEllAR-GROUP/hpx/pull/4421
1106 https://github.com/STEllAR-GROUP/hpx/pull/4417
1107 https://github.com/STEllAR-GROUP/hpx/pull/4415
1108 https://github.com/STEllAR-GROUP/hpx/pull/4414
1109 https://github.com/STEllAR-GROUP/hpx/pull/4413
1110 https://github.com/STEllAR-GROUP/hpx/pull/4412
1111 https://github.com/STEllAR-GROUP/hpx/pull/4411
1112 https://github.com/STEllAR-GROUP/hpx/pull/4407
1113 https://github.com/STEllAR-GROUP/hpx/pull/4405
1114 https://github.com/STEllAR-GROUP/hpx/pull/4404
1115 https://github.com/STEllAR-GROUP/hpx/pull/4399
1116 https://github.com/STEllAR-GROUP/hpx/pull/4398
1117 https://github.com/STEllAR-GROUP/hpx/pull/4395
1118 https://github.com/STEllAR-GROUP/hpx/pull/4393
1119 https://github.com/STEllAR-GROUP/hpx/pull/4391
1120 https://github.com/STEllAR-GROUP/hpx/pull/4390
1121 https://github.com/STEllAR-GROUP/hpx/pull/4388
1122 https://github.com/STEllAR-GROUP/hpx/pull/4387
• PR #4386 - Add warning messages for using thread binding options on macOS
• PR #4385 - Cuda futures
• PR #4384 - Make enabling dynamic hpx_main on non-Linux systems a configuration error
• PR #4383 - Use configure_file for HPXCacheVariables.cmake
• PR #4382 - Update spellchecking whitelist and fix more typos
• PR #4380 - Add a helper function to get a future from a cuda stream
• PR #4379 - Add Windows and macOS CI with GitHub actions
• PR #4378 - Change C++ standard handling
• PR #4377 - Remove Python scripts
• PR #4374 - Adding overload for hpx::init/hpx::start for use with resource partitioner
• PR #4373 - Adding test that verifies for 4369 to be fixed
• PR #4372 - Another attempt at fixing the integral mismatch and conversion warnings
• PR #4370 - Doc updates quick start
• PR #4368 - Add a whitelist of words for weird spelling suggestions
• PR #4366 - Suppress or fix clang-tidy-9 warnings
• PR #4365 - Removing more Boost dependencies
• PR #4363 - Update clang-format config file for version 9
• PR #4362 - Fix indices typo
• PR #4361 - Boost cleanup
• PR #4360 - Move plugins
• PR #4358 - Doc updates; generating documentation. Will likely need heavy editing.
• PR #4356 - Remove some minor unused and unnecessary Boost includes
• PR #4355 - Fix spellcheck step in CircleCI config

1123 https://github.com/STEllAR-GROUP/hpx/pull/4386
1124 https://github.com/STEllAR-GROUP/hpx/pull/4385
1125 https://github.com/STEllAR-GROUP/hpx/pull/4384
1126 https://github.com/STEllAR-GROUP/hpx/pull/4383
1127 https://github.com/STEllAR-GROUP/hpx/pull/4382
1128 https://github.com/STEllAR-GROUP/hpx/pull/4380
1129 https://github.com/STEllAR-GROUP/hpx/pull/4379
1130 https://github.com/STEllAR-GROUP/hpx/pull/4378
1131 https://github.com/STEllAR-GROUP/hpx/pull/4377
1132 https://github.com/STEllAR-GROUP/hpx/pull/4374
1133 https://github.com/STEllAR-GROUP/hpx/pull/4373
1134 https://github.com/STEllAR-GROUP/hpx/pull/4372
1135 https://github.com/STEllAR-GROUP/hpx/pull/4370
1136 https://github.com/STEllAR-GROUP/hpx/pull/4368
1137 https://github.com/STEllAR-GROUP/hpx/pull/4366
1138 https://github.com/STEllAR-GROUP/hpx/pull/4365
1139 https://github.com/STEllAR-GROUP/hpx/pull/4363
1140 https://github.com/STEllAR-GROUP/hpx/pull/4362
1141 https://github.com/STEllAR-GROUP/hpx/pull/4361
1142 https://github.com/STEllAR-GROUP/hpx/pull/4360
1143 https://github.com/STEllAR-GROUP/hpx/pull/4358
1144 https://github.com/STEllAR-GROUP/hpx/pull/4356
1145 https://github.com/STEllAR-GROUP/hpx/pull/4355
Chapter 2. What’s so special about HPX?

- PR #4354 - Lightweight utility to hold a pack as members
- PR #4352 - Minor fixes to the C++ standard detection for MSVC
- PR #4351 - Move generated documentation to hpx-docs repo
- PR #4347 - Add cmake policy - CMP0074
- PR #4346 - Remove file committed by mistake
- PR #4342 - Remove HCC and SYCL options from CMakeLists.txt
- PR #4341 - Fix launch process test with APEX enabled
- PR #4340 - Testing Cirrus CI
- PR #4339 - Post 1.4.0 updates
- PR #4338 - Spelling corrections and CircleCI spell check
- PR #4333 - Flatten bound callables
- PR #4332 - This is a collection of mostly minor (cleanup) fixes
- PR #4331 - This adds the missing tests for async_colocated and async_continue_colocated
- PR #4330 - Remove HPX.Compute host default_executor
- PR #4329 - Generate global header for basic_execution module
- PR #4327 - Use INTERNAL_FLAGS option for all examples and components
- PR #4326 - Usage of temporary allocator in assignment operator of compute::vector
- PR #4325 - Use hpx::threads::get_cache_line_size in prefetching.hpp
- PR #4324 - Enable compatibility headers option for execution module
- PR #4316 - Add clang format indentppdirectives
- PR #4313 - Introduce index_pack alias to pack of size_t
- PR #4312 - Fixing compatibility header for pack.hpp
- PR #4311 - Dataflow annotations for APEX

1146 https://github.com/STEllAR-GROUP/hpx/pull/4354
1147 https://github.com/STEllAR-GROUP/hpx/pull/4352
1148 https://github.com/STEllAR-GROUP/hpx/pull/4351
1149 https://github.com/STEllAR-GROUP/hpx/pull/4347
1150 https://github.com/STEllAR-GROUP/hpx/pull/4346
1151 https://github.com/STEllAR-GROUP/hpx/pull/4342
1152 https://github.com/STEllAR-GROUP/hpx/pull/4341
1153 https://github.com/STEllAR-GROUP/hpx/pull/4340
1154 https://github.com/STEllAR-GROUP/hpx/pull/4339
1155 https://github.com/STEllAR-GROUP/hpx/pull/4338
1156 https://github.com/STEllAR-GROUP/hpx/pull/4333
1157 https://github.com/STEllAR-GROUP/hpx/pull/4332
1158 https://github.com/STEllAR-GROUP/hpx/pull/4331
1159 https://github.com/STEllAR-GROUP/hpx/pull/4330
1160 https://github.com/STEllAR-GROUP/hpx/pull/4328
1161 https://github.com/STEllAR-GROUP/hpx/pull/4327
1162 https://github.com/STEllAR-GROUP/hpx/pull/4326
1163 https://github.com/STEllAR-GROUP/hpx/pull/4325
1164 https://github.com/STEllAR-GROUP/hpx/pull/4324
1165 https://github.com/STEllAR-GROUP/hpx/pull/4316
1166 https://github.com/STEllAR-GROUP/hpx/pull/4313
1167 https://github.com/STEllAR-GROUP/hpx/pull/4312
1168 https://github.com/STEllAR-GROUP/hpx/pull/4311
2.10. Releases

- PR #4309\(^{1169}\) - Update launching_and_configuring_hpx_applications.rst
- PR #4306\(^{1170}\) - Fix schedule hint not being taken from executor
- PR #4305\(^{1171}\) - Implementing `hpx::functional::tag_invoke`
- PR #4304\(^{1172}\) - Improve pack support utilities
- PR #4303\(^{1173}\) - Remove errors module dependency on datastructures
- PR #4301\(^{1174}\) - Clean up thread executors
- PR #4294\(^{1175}\) - Logging revamp
- PR #4292\(^{1176}\) - Remove SPDX tag from Boost License file to allow for github to recognize it
- PR #4291\(^{1177}\) - Add format support for std::tm
- PR #4290\(^{1178}\) - Simplify compatible tuples check
- PR #4288\(^{1179}\) - A lightweight take on boost::lexical_cast
- PR #4287\(^{1180}\) - Forking boost::lexical_cast as a new module
- PR #4277\(^{1181}\) - MPI_futures
- PR #4270\(^{1182}\) - Refactor future implementation
- PR #4265\(^{1183}\) - Threading module
- PR #4259\(^{1184}\) - Module naming base
- PR #4251\(^{1185}\) - Local workrequesting scheduler
- PR #4250\(^{1186}\) - Inline execution of scoped tasks, if possible
- PR #4247\(^{1187}\) - Add execution in module headers
- PR #4246\(^{1188}\) - Expose CMake targets officially
- PR #4239\(^{1189}\) - Doc updates miscellaneous (partially completed during Google Season of Docs)
- PR #4233\(^{1190}\) - Remove project() from modules + fix CMAKE_SOURCE_DIR issue
- PR #4231\(^{1191}\) - Module local lcos

\(^{1169}\) https://github.com/STEllAR-GROUP/hpx/pull/4309
\(^{1170}\) https://github.com/STEllAR-GROUP/hpx/pull/4306
\(^{1171}\) https://github.com/STEllAR-GROUP/hpx/pull/4305
\(^{1172}\) https://github.com/STEllAR-GROUP/hpx/pull/4304
\(^{1173}\) https://github.com/STEllAR-GROUP/hpx/pull/4303
\(^{1174}\) https://github.com/STEllAR-GROUP/hpx/pull/4301
\(^{1175}\) https://github.com/STEllAR-GROUP/hpx/pull/4294
\(^{1176}\) https://github.com/STEllAR-GROUP/hpx/pull/4292
\(^{1177}\) https://github.com/STEllAR-GROUP/hpx/pull/4291
\(^{1178}\) https://github.com/STEllAR-GROUP/hpx/pull/4290
\(^{1179}\) https://github.com/STEllAR-GROUP/hpx/pull/4288
\(^{1180}\) https://github.com/STEllAR-GROUP/hpx/pull/4287
\(^{1181}\) https://github.com/STEllAR-GROUP/hpx/pull/4277
\(^{1182}\) https://github.com/STEllAR-GROUP/hpx/pull/4270
\(^{1183}\) https://github.com/STEllAR-GROUP/hpx/pull/4265
\(^{1184}\) https://github.com/STEllAR-GROUP/hpx/pull/4259
\(^{1185}\) https://github.com/STEllAR-GROUP/hpx/pull/4251
\(^{1186}\) https://github.com/STEllAR-GROUP/hpx/pull/4250
\(^{1187}\) https://github.com/STEllAR-GROUP/hpx/pull/4247
\(^{1188}\) https://github.com/STEllAR-GROUP/hpx/pull/4246
\(^{1189}\) https://github.com/STEllAR-GROUP/hpx/pull/4239
\(^{1190}\) https://github.com/STEllAR-GROUP/hpx/pull/4233
\(^{1191}\) https://github.com/STEllAR-GROUP/hpx/pull/4231
2.10.6 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 - HPX 1.4.0 does not compile with gcc 10
- Issue #4336 - Building HPX 1.4.0 with IO Counters breaks (Windows)
- Issue #4334 - HPX Debug and RelWithDebinfo builds on Windows not installing .pdb files
- Issue #4322 - Undefine VT1 and VT2 after boost includes
- Issue #4314 - Compile error on 1.4.0
- Issue #4307 - ld: error: duplicate symbol: freebsd_environ

1192 https://github.com/STEllAR-GROUP/hpx/pull/4207
1193 https://github.com/STEllAR-GROUP/hpx/pull/4206
1194 https://github.com/STEllAR-GROUP/hpx/pull/4141
1195 https://github.com/STEllAR-GROUP/hpx/pull/4091
1196 https://github.com/STEllAR-GROUP/hpx/pull/4017
1197 https://github.com/STEllAR-GROUP/hpx/issues/4320
1198 https://github.com/STEllAR-GROUP/hpx/issues/4336
1199 https://github.com/STEllAR-GROUP/hpx/issues/4334
1200 https://github.com/STEllAR-GROUP/hpx/issues/4322
1201 https://github.com/STEllAR-GROUP/hpx/issues/4314
1202 https://github.com/STEllAR-GROUP/hpx/issues/4307
Closed pull requests

- PR #4376 - Attempt to fix some test build errors on Windows
- PR #4357 - Adding missing #includes to fix gcc V10 linker problems
- PR #4353 - Skip MPI_Finalize if MPI_Init is not called from HPX
- PR #4343 - Give a hard error if IO counters are enabled on non-Linux systems
- PR #4377 - Installing pdb files on Windows
- PR #4335 - Adding capability to buildsystem to use an installed version of HPX
- PR #4315 - Forcing exported symbols from composable_guard to be linked into core library
- PR #4310 - Remove environment handling from exception.cpp

2.10.7 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

1203 https://github.com/STEllAR-GROUP/hpx/pull/4376
1204 https://github.com/STEllAR-GROUP/hpx/pull/4357
1205 https://github.com/STEllAR-GROUP/hpx/pull/4353
1206 https://github.com/STEllAR-GROUP/hpx/pull/4343
1207 https://github.com/STEllAR-GROUP/hpx/pull/4337
1208 https://github.com/STEllAR-GROUP/hpx/pull/4335
1209 https://github.com/STEllAR-GROUP/hpx/pull/4315
1210 https://github.com/STEllAR-GROUP/hpx/pull/4310
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 en-
vvironments 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.
- 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_COMPATIBILITY option and the associated compatibility layer has been re-
  moved.

Closed issues

- Issue #42821211 - Build Issues with Release on Windows
- Issue #42781212 - Build Issues with CMake 3.14.4
- Issue #42731213 - Clients of HPX 1.4.0-rc2 with APEX ar not linked to libhpx-apex

1211 https://github.com/STEllAR-GROUP/hpx/issues/4282
1212 https://github.com/STEllAR-GROUP/hpx/issues/4278
1213 https://github.com/STEllAR-GROUP/hpx/issues/4273
• Issue #4269 - Building HPX 1.4.0-rc2 with support for APEX fails
• Issue #4263 - Compilation fail on latest master
• Issue #4232 - Configure of HPX project using CMake FetchContent fails
• Issue #4223 - “Re-using the main() function as the main HPX entry point” doesn’t work
• Issue #4220 - HPX won’t compile - error building resource_partitioner
• Issue #4215 - HPX 1.4.0rc1 does not link on s390x
• Issue #4204 - Trouble compiling HPX with Intel compiler
• Issue #4199 - Refactor APEX to eliminate circular dependency
• Issue #4187 - HPX can’t build on OSX
• Issue #4185 - Simple debug output for development
• Issue #4182 - @HPX_CONF_PREFIX@ is the empty string
• Issue #4169 - HPX won’t build with APEX
• Issue #4163 - Add back HPX_LIBRARIES and HPX_INCLUDE_DIRS
• Issue #4161 - It should be possible to call find_package(HPX) multiple times
• Issue #4155 - get_self_id() for stackless threads returns invalid_thread_id
• Issue #4151 - build error with MPI code
• Issue #4150 - hpx won’t build on POWER9 with clang 8
• Issue #4148 - cacheline_data delivers poor performance with C++17 compared to C++14
• Issue #4144 - target general in HPX_LIBRARIES does not exist
• Issue #4134 - CMake Error when -DHPX_WITH_HPXMP=ON
• Issue #4132 - parallel fill leaves elements unfilled
• Issue #4123 - PAPI performance counters are inaccessible
• Issue #4118 - static_chunk_size is not obeyed in scan algorithms

https://github.com/STEllAR-GROUP/hpx/issues/4269
https://github.com/STEllAR-GROUP/hpx/issues/4263
https://github.com/STEllAR-GROUP/hpx/issues/4232
https://github.com/STEllAR-GROUP/hpx/issues/4223
https://github.com/STEllAR-GROUP/hpx/issues/4220
https://github.com/STEllAR-GROUP/hpx/issues/4215
https://github.com/STEllAR-GROUP/hpx/issues/4204
https://github.com/STEllAR-GROUP/hpx/issues/4199
https://github.com/STEllAR-GROUP/hpx/issues/4187
https://github.com/STEllAR-GROUP/hpx/issues/4185
https://github.com/STEllAR-GROUP/hpx/issues/4182
https://github.com/STEllAR-GROUP/hpx/issues/4169
https://github.com/STEllAR-GROUP/hpx/issues/4163
https://github.com/STEllAR-GROUP/hpx/issues/4161
https://github.com/STEllAR-GROUP/hpx/issues/4155
https://github.com/STEllAR-GROUP/hpx/issues/4151
https://github.com/STEllAR-GROUP/hpx/issues/4150
https://github.com/STEllAR-GROUP/hpx/issues/4148
https://github.com/STEllAR-GROUP/hpx/issues/4144
https://github.com/STEllAR-GROUP/hpx/issues/4134
https://github.com/STEllAR-GROUP/hpx/issues/4132
https://github.com/STEllAR-GROUP/hpx/issues/4123
https://github.com/STEllAR-GROUP/hpx/issues/4118
• Issue #4115 - dependency chaining error with APEX
• Issue #4107 - Initializing runtime without entry point function and command line arguments
• Issue #4105 - Bug in `hpx:bind=numa-balanced`
• Issue #4101 - Bound tasks
• Issue #4100 - Add SPDX identifier to all files
• Issue #4085 - `hpx_topology` library should depend on hwloc
• Issue #4067 - HPX fails to build on macOS
• Issue #4056 - Building without thread manager idle backoff fails
• Issue #4052 - Enforce `clang-format` style for modules
• Issue #4032 - Simple hello world fails to launch correctly
• Issue #4030 - Allow threads to skip context switching
• Issue #4029 - Add support for mimalloc
• Issue #4005 - Can’t link HPX when APEX enabled
• Issue #4002 - Missing header for algorithm module
• Issue #3989 - conversion from `long` to `unsigned int` requires a narrowing conversion on MSVC
• Issue #3958 - `/statistics/average@` perf counter can’t be created
• Issue #3953 - CMake errors from `HPX_AddPseudoDependencies`
• Issue #3941 - CMake error for APEX install target
• Issue #3940 - Convert pseudo-doxygen function documentation into actual doxygen documentation
• Issue #3935 - HPX compiler match too strict?
• Issue #3929 - Buildbot failures on latest HPX stable
• Issue #3912 - I recommend publishing a version that does not depend on the boost library
• Issue #3890 - `hpx.ini` not working

1237 https://github.com/STEllAR-GROUP/hpx/issues/4115
1238 https://github.com/STEllAR-GROUP/hpx/issues/4107
1239 https://github.com/STEllAR-GROUP/hpx/issues/4105
1240 https://github.com/STEllAR-GROUP/hpx/issues/4101
1241 https://github.com/STEllAR-GROUP/hpx/issues/4100
1242 https://github.com/STEllAR-GROUP/hpx/issues/4085
1243 https://github.com/STEllAR-GROUP/hpx/issues/4067
1244 https://github.com/STEllAR-GROUP/hpx/issues/4056
1245 https://github.com/STEllAR-GROUP/hpx/issues/4052
1246 https://github.com/STEllAR-GROUP/hpx/issues/4032
1247 https://github.com/STEllAR-GROUP/hpx/issues/4030
1248 https://github.com/STEllAR-GROUP/hpx/issues/4029
1249 https://github.com/STEllAR-GROUP/hpx/issues/4005
1250 https://github.com/STEllAR-GROUP/hpx/issues/4002
1251 https://github.com/STEllAR-GROUP/hpx/issues/3989
1252 https://github.com/STEllAR-GROUP/hpx/issues/3958
1253 https://github.com/STEllAR-GROUP/hpx/issues/3953
1254 https://github.com/STEllAR-GROUP/hpx/issues/3941
1255 https://github.com/STEllAR-GROUP/hpx/issues/3940
1256 https://github.com/STEllAR-GROUP/hpx/issues/3935
1257 https://github.com/STEllAR-GROUP/hpx/issues/3929
1258 https://github.com/STEllAR-GROUP/hpx/issues/3912
1259 https://github.com/STEllAR-GROUP/hpx/issues/3890

Chapter 2. What’s so special about **HPX**?
• Issue #3883 - cuda compilation fails because of `-faligned-new`
• Issue #3879 - HPX fails to configure with `-DPX_WITH_TESTS=OFF`
• Issue #3871 - dataflow does not support void allocators
• Issue #3867 - Latest HTML docs placed in wrong directory on GitHub pages
• Issue #3866 - Make sure all tests use `HPX_TEST*` macros and not `HPX_ASSERT`
• Issue #3857 - CMake all-keyword or all-plain for `target_link_libraries`
• Issue #3856 - `hpx_setup_target` adds rogue flags
• Issue #3850 - HPX fails to build on POWER8 with Clang7
• Issue #3848 - Remove lva member from `thread_init_data`
• Issue #3838 - `hpx::parallel::count/count_if` failing tests
• Issue #3651 - `hpx::parallel::transform_reduce` with non const reference as lambda parameter
• Issue #3560 - Apex integration with HPX not working properly
• Issue #3322 - No warning when mixing debug/release builds

Closed pull requests

• PR #4300 - Checks for `MPI_Init` being called twice
• PR #4299 - Small CMake fixes
• PR #4298 - Remove extra call to annotate function that messes up traces
• PR #4296 - Fixing collectives locking problem
• PR #4295 - Do not check `LICENSE_1_0.txt` for inspect violations
• PR #4293 - Applying two small changes fixing carious MSVC/Windows problems
• PR #4285 - Delete `apex.hpp`
• PR #4276 - Disable doxygen generation for `hpx/debugging/print.hpp` file
• PR #4275 - Make sure APEX is linked to even when not explicitly referenced
• PR #4272 - Fix pushing of documentation
• PR #4271 - Updating APEX tag, don’t create new task_wrapper on operator= of hpx_thread object
• PR #4268 - Testing for noexcept function specializations in C++11/14 mode
• PR #4267 - Fixing MSVC warning
• PR #4266 - Make sure macOS Travis CI fails if build step fails
• PR #4264 - Clean up compatibility header options
• PR #4262 - Cleanup modules CMakeLists.txt
• PR #4261 - Fixing HPX/APEX linking and dependencies for external projects like Phylanx
• PR #4260 - Fix docs compilation problems
• PR #4258 - Couple of minor changes
• PR #4257 - Fix apex annotation for async dispatch
• PR #4256 - Remove lambdas from assert expressions
• PR #4255 - Ignoring lock in all_to_all and all_reduce
• PR #4254 - Adding action specializations for noexcept functions
• PR #4253 - Move partlit.hpp to affinity module
• PR #4252 - Make mismatching build types a hard error in CMake
• PR #4249 - Scheduler improvement
• PR #4248 - update hpxmp tag to v0.3.0
• PR #4245 - Adding high performance channels
• PR #4244 - Ignore lock in ignore_while_locked_1485 test
• PR #4243 - Fix PAPI command line option documentation
• PR #4242 - Ignore lock in target_distribution_policy
• PR #4241 - Fix start_stop_callbacks test

1282 https://github.com/STEllAR-GROUP/hpx/pull/4272
1283 https://github.com/STEllAR-GROUP/hpx/pull/4271
1284 https://github.com/STEllAR-GROUP/hpx/pull/4268
1285 https://github.com/STEllAR-GROUP/hpx/pull/4267
1286 https://github.com/STEllAR-GROUP/hpx/pull/4266
1287 https://github.com/STEllAR-GROUP/hpx/pull/4264
1288 https://github.com/STEllAR-GROUP/hpx/pull/4262
1289 https://github.com/STEllAR-GROUP/hpx/pull/4261
1290 https://github.com/STEllAR-GROUP/hpx/pull/4260
1291 https://github.com/STEllAR-GROUP/hpx/pull/4258
1292 https://github.com/STEllAR-GROUP/hpx/pull/4257
1293 https://github.com/STEllAR-GROUP/hpx/pull/4256
1294 https://github.com/STEllAR-GROUP/hpx/pull/4255
1295 https://github.com/STEllAR-GROUP/hpx/pull/4254
1296 https://github.com/STEllAR-GROUP/hpx/pull/4253
1297 https://github.com/STEllAR-GROUP/hpx/pull/4252
1298 https://github.com/STEllAR-GROUP/hpx/pull/4249
1299 https://github.com/STEllAR-GROUP/hpx/pull/4248
1300 https://github.com/STEllAR-GROUP/hpx/pull/4245
1301 https://github.com/STEllAR-GROUP/hpx/pull/4244
1302 https://github.com/STEllAR-GROUP/hpx/pull/4243
1303 https://github.com/STEllAR-GROUP/hpx/pull/4242
1304 https://github.com/STEllAR-GROUP/hpx/pull/4241
• PR #4240 - Mostly fix clang CUDA compilation
• PR #4238 - Google Season of Docs updates to documentation; grammar edits.
• PR #4237 - fixing annotated task to use the name, not the desc
• PR #4236 - Move module print summary to modules
• PR #4235 - Don’t use alignas in cache\{aligned, line\}_data
• PR #4234 - Add basic overview sentence to all modules
• PR #4230 - Add OS X builds to Travis CI
• PR #4229 - Remove leftover queue compatibility checks
• PR #4226 - Fixing APEX shutdown by explicitly shutting down throttling
• PR #4225 - Allow CMAKE_INSTALL_PREFIX to be a relative path
• PR #4224 - Deprecate verbs parcelport
• PR #4222 - Update register\{thread, work\} namespaces
• PR #4221 - Changing HPX\_GCC\_VERSION check from 70000 to 70300
• PR #4218 - Google Season of Docs updates to documentation; grammar edits.
• PR #4217 - Google Season of Docs updates to documentation; grammar edits.
• PR #4216 - Fixing gcc warning on 32bit platforms (integer truncation)
• PR #4214 - Apex callback refactoring
• PR #4213 - Clean up allocator checks for dependent projects
• PR #4212 - Google Season of Docs updates to documentation; grammar edits.
• PR #4211 - Google Season of Docs updates to documentation; contributing to hpx
• PR #4210 - Attempting to fix Intel compilation
• PR #4209 - Fix CUDA 10 build
• PR #4205 - Making sure that differences in CMAKE\_BUILD\_TYPE are not reported on multi-configuration cmake generators

https://github.com/STEllAR-GROUP/hpx/pull/4240
https://github.com/STEllAR-GROUP/hpx/pull/4238
https://github.com/STEllAR-GROUP/hpx/pull/4237
https://github.com/STEllAR-GROUP/hpx/pull/4236
https://github.com/STEllAR-GROUP/hpx/pull/4235
https://github.com/STEllAR-GROUP/hpx/pull/4234
https://github.com/STEllAR-GROUP/hpx/pull/4230
https://github.com/STEllAR-GROUP/hpx/pull/4229
https://github.com/STEllAR-GROUP/hpx/pull/4226
https://github.com/STEllAR-GROUP/hpx/pull/4225
https://github.com/STEllAR-GROUP/hpx/pull/4224
https://github.com/STEllAR-GROUP/hpx/pull/4222
https://github.com/STEllAR-GROUP/hpx/pull/4221
https://github.com/STEllAR-GROUP/hpx/pull/4218
https://github.com/STEllAR-GROUP/hpx/pull/4217
https://github.com/STEllAR-GROUP/hpx/pull/4216
https://github.com/STEllAR-GROUP/hpx/pull/4214
https://github.com/STEllAR-GROUP/hpx/pull/4213
https://github.com/STEllAR-GROUP/hpx/pull/4212
https://github.com/STEllAR-GROUP/hpx/pull/4211
https://github.com/STEllAR-GROUP/hpx/pull/4210
https://github.com/STEllAR-GROUP/hpx/pull/4209
https://github.com/STEllAR-GROUP/hpx/pull/4205
• PR #4203 - Deprecate Vc
• PR #4202 - Fix CUDA configuration
• PR #4200 - Making sure hpx_wrap is not passed on to linker on non-Linux systems
• PR #4198 - Fix execution_agent.cpp compilation with GCC 5
• PR #4197 - Remove deprecated options for 1.4.0 release
• PR #4196 - minor fixes for building on OSX Darwin
• PR #4195 - Use full clone on CircleCI for pushing stable tag
• PR #4193 - Add scheduling hints to hello_world_distributed
• PR #4192 - Set up CUDA in HPXConfig.cmake
• PR #4191 - Export allocators root variables
• PR #4190 - Don’t use constexpr in thread_data with GCC <= 6
• PR #4189 - Only use quick_exit if available
• PR #4188 - Google Season of Docs updates to documentation; writing single node hpx applications
• PR #4186 - correct vc to cuda in cuda cmake
• PR #4184 - Resetting some cached variables to make sure those are re-filled
• PR #4183 - Fix hpxcxx configuration
• PR #4181 - Rename base libraries var
• PR #4180 - Move header left behind earlier to plugin module
• PR #4179 - Moving zip_iterator and transform_iterator to iterator_support module
• PR #4178 - Move checkpointing support to its own module
• PR #4177 - Small const fix to basic_execution module
• PR #4176 - Add back HPX_LIBRARIES and friends to HPXConfig.cmake
• PR #4175 - Make Vc public and add it to HPXConfig.cmake
- PR #4173 - Wait for runtime to be running before returning from hpx::start
- PR #4172 - More protection against shutdown problems in error handling scenarios.
- PR #4171 - Ignore lock in condition_variable::wait
- PR #4170 - Adding APEX dependency to MPI parcelport
- PR #4168 - Adding utility include
- PR #4167 - Add a condition to setup the external libraries
- PR #4166 - Add an INTERNAL_FLAGS option to link to hpx_internal_flags
- PR #4165 - Forward HPX_* cmake cache variables to external projects
- PR #4164 - Affinity and batch environment modules
- PR #4162 - Handle quick exit
- PR #4160 - Using target_link_libraries for cmake versions >= 3.12
- PR #4159 - Make sure HPX_WITH_NATIVE_TLS is forwarded to dependent projects
- PR #4158 - Adding allocator imported target as a dependency of allocator module
- PR #4157 - Add hpx_memory as a dependency of parcelport plugins
- PR #4156 - Stackless coroutines now can refer to themselves (through get_self() and friends)
- PR #4154 - Added CMake policy CMP0060 for HPX applications.
- PR #4153 - add header iomanip to tests and tool
- PR #4152 - Casting MPI tag value
- PR #4149 - Add back private m_desc member variable in program_options module
- PR #4147 - Resource partitioner and threadmanager modules
- PR #4146 - Google Season of Docs updates to documentation; creating hpx projects
- PR #4145 - Adding basic support for stackless threads
- PR #4143 - Exclude test_client_1950 from all target
• PR #4142 Add a new thread_pool_executor
• PR #4146 - Google Season of Docs updates to documentation; why hpx
• PR #4139 - Remove runtime includes from coroutines module
• PR #4138 - Forking boost::intrusive_ptr and adding it as hpx::intrusive_ptr
• PR #4137 - Fixing TSS destruction
• PR #4136 - HPX.Compute modules
• PR #4133 - Fix block_executor
• PR #4131 - Applying fixes based on reports from PVS Studio
• PR #4130 - Adding missing header to build system
• PR #4129 - Fixing compilation if HPX_WITH_DATAPAR_VC is enabled
• PR #4128 - Renaming moveonly_any to unique_any
• PR #4126 - Attempt to fix basic_any constructor for gcc 7
• PR #4125 - Changing extra_archive_data implementation
• PR #4124 - Don’t link to Boost.System unless required
• PR #4122 - Add kernel launch helper utility (+saxpy demo) and merge in octotiger changes
• PR #4121 - Fixing migration test if networking is disabled.
• PR #4120 - Google Season of Docs updates to documentation; hpx build system v1
• PR #4119 - Making sure chunk_size and max_chunk are actually applied to parallel algorithms if specified
• PR #4117 - Make CircleCI formatting check store diff
• PR #4116 - Fix automatically setting C++ standard
• PR #4114 - Module serialization
• PR #4113 - Module datastructures
• PR #4111 - Fixing performance regression introduced earlier
- PR #4110 - Adding missing SPDX tags
- PR #4108 - Overload for start without entry point/argv.
- PR #4108 - Making sure C++ standard is properly detected and propagated
- PR #4106 - Use std::round for guaranteed rounding without errors
- PR #4104 - Extend scheduler_mode with new work_stealing and task assignment modes
- PR #4103 - Add this to lambda capture list
- PR #4102 - Add spdx license and check
- PR #4099 - Module coroutines
- PR #4098 - Fix append module path in module CMakeLists template
- PR #4097 - Function tests
- PR #4096 - Removing return of thread_result_type from functions not needing them
- PR #4095 - Stop-gap measure until cmake overhaul is in place
- PR #4094 - Deprecate HPX_WITH_MORE_THAN_64_THREADS
- PR #4093 - Fix initialization of global_num_tasks in parallel_executor
- PR #4092 - Add support for mi-malloc
- PR #4091 - Execution context
- PR #4090 - Make counters in coroutines optional
- PR #4087 - Making hpx::util::any compatible with C++17
- PR #4084 - Making sure destination array for std::transform is properly resized
- PR #4083 - Adapting thread_queue_mc to behave even if no 128bit atomics are available
- PR #4082 - Fix compilation on GCC 5
- PR #4081 - Adding option allowing to force using Boost.FileSystem
- PR #4080 - Updating module dependencies

- https://github.com/STEllAR-GROUP/hpx/pull/4110
- https://github.com/STEllAR-GROUP/hpx/pull/4109
- https://github.com/STEllAR-GROUP/hpx/pull/4108
- https://github.com/STEllAR-GROUP/hpx/pull/4106
- https://github.com/STEllAR-GROUP/hpx/pull/4104
- https://github.com/STEllAR-GROUP/hpx/pull/4103
- https://github.com/STEllAR-GROUP/hpx/pull/4102
- https://github.com/STEllAR-GROUP/hpx/pull/4099
- https://github.com/STEllAR-GROUP/hpx/pull/4098
- https://github.com/STEllAR-GROUP/hpx/pull/4097
- https://github.com/STEllAR-GROUP/hpx/pull/4096
- https://github.com/STEllAR-GROUP/hpx/pull/4095
- https://github.com/STEllAR-GROUP/hpx/pull/4094
- https://github.com/STEllAR-GROUP/hpx/pull/4093
- https://github.com/STEllAR-GROUP/hpx/pull/4092
- https://github.com/STEllAR-GROUP/hpx/pull/4091
- https://github.com/STEllAR-GROUP/hpx/pull/4090
- https://github.com/STEllAR-GROUP/hpx/pull/4089
- https://github.com/STEllAR-GROUP/hpx/pull/4088
- https://github.com/STEllAR-GROUP/hpx/pull/4087
- https://github.com/STEllAR-GROUP/hpx/pull/4086
- https://github.com/STEllAR-GROUP/hpx/pull/4085
- https://github.com/STEllAR-GROUP/hpx/pull/4084
- https://github.com/STEllAR-GROUP/hpx/pull/4083
- https://github.com/STEllAR-GROUP/hpx/pull/4082
- https://github.com/STEllAR-GROUP/hpx/pull/4081
- https://github.com/STEllAR-GROUP/hpx/pull/4080
• PR #4079 - Add missing tests for iterator_support module
• PR #4078 - Disable parcel-layer if networking is disabled
• PR #4077 - Add missing include that causes build fails
• PR #4076 - Enable compatibility headers for functional module
• PR #4075 - Coroutines module
• PR #4073 - Use configure_file for generated files in modules
• PR #4071 - Fixing MPI detection for PMIx
• PR #4070 - Fix macOS builds
• PR #4069 - Moving more facilities to the collectives module
• PR #4068 - Adding main HPX #include directory to modules
• PR #4066 - Switching the use of message(STATUS "...") to hpx_info
• PR #4065 - Move Boost.Filesystem handling to filesystem module
• PR #4064 - Fix program_options test with older boost versions
• PR #4062 - The cpu_features tool fails to compile on anything but x86 architectures
• PR #4061 - Add clang-format checking step for modules
• PR #4060 - Making sure HPX_IDLE_BACKOFF_TIME_MIN is always defined (even if its unused)
• PR #4059 - Renaming module hpx_parallel_executors into hpx_execution
• PR #4058 - Do not build networking tests when networking disabled
• PR #4057 - Printing configuration summary for modules as well
• PR #4055 - Google Season of Docs updates to documentation; hpx build systems
• PR #4054 - Add troubleshooting section to manual
• PR #4051 - Add more variations to future_overhead test
• PR #4050 - Creating plugin module

https://github.com/STEllAR-GROUP/hpx/pull/4079
https://github.com/STEllAR-GROUP/hpx/pull/4078
https://github.com/STEllAR-GROUP/hpx/pull/4077
https://github.com/STEllAR-GROUP/hpx/pull/4076
https://github.com/STEllAR-GROUP/hpx/pull/4075
https://github.com/STEllAR-GROUP/hpx/pull/4073
https://github.com/STEllAR-GROUP/hpx/pull/4071
https://github.com/STEllAR-GROUP/hpx/pull/4070
https://github.com/STEllAR-GROUP/hpx/pull/4069
https://github.com/STEllAR-GROUP/hpx/pull/4068
https://github.com/STEllAR-GROUP/hpx/pull/4066
https://github.com/STEllAR-GROUP/hpx/pull/4065
https://github.com/STEllAR-GROUP/hpx/pull/4064
https://github.com/STEllAR-GROUP/hpx/pull/4062
https://github.com/STEllAR-GROUP/hpx/pull/4061
https://github.com/STEllAR-GROUP/hpx/pull/4060
https://github.com/STEllAR-GROUP/hpx/pull/4059
https://github.com/STEllAR-GROUP/hpx/pull/4058
https://github.com/STEllAR-GROUP/hpx/pull/4057
https://github.com/STEllAR-GROUP/hpx/pull/4055
https://github.com/STEllAR-GROUP/hpx/pull/4054
https://github.com/STEllAR-GROUP/hpx/pull/4051
https://github.com/STEllAR-GROUP/hpx/pull/4050
- PR #4049 - Move missing modules tests
- PR #4047 - Add boost/filesystem headers to inspect deprecated headers
- PR #4045 - Module functional
- PR #4043 - Fix preconditions and error messages for suspension functions
- PR #4041 - Pass HPX_STANDARD on to dependent projects via HPXConfig.cmake
- PR #4040 - Program options module
- PR #4039 - Moving non-serializable any (any_nonser) to datastructures module
- PR #4038 - Adding MPark’s variant (V1.4.0) to HPX
- PR #4037 - Adding resiliency module
- PR #4036 - Add C++17 filesystem compatibility header
- PR #4035 - Fixing support for mpirun
- PR #4028 - CMake to target based directives
- PR #4027 - Remove GitLab CI configuration
- PR #4026 - Threading refactoring
- PR #4025 - Refactoring thread queue configuration options
- PR #4024 - Fix padding calculation in cache_aligned_data.hpp
- PR #4023 - Fixing Codacy issues
- PR #4022 - Make sure process mask option is passed to affinity_data
- PR #4021 - Warn about compiling in C++11 mode
- PR #4020 - Module concurrency
- PR #4019 - Module topology
- PR #4018 - Update deprecated header in thread_queue_mc.hpp
- PR #4015 - Avoid overwriting artifacts

https://github.com/STEllAR-GROUP/hpx/pull/4049
https://github.com/STEllAR-GROUP/hpx/pull/4047
https://github.com/STEllAR-GROUP/hpx/pull/4045
https://github.com/STEllAR-GROUP/hpx/pull/4043
https://github.com/STEllAR-GROUP/hpx/pull/4041
https://github.com/STEllAR-GROUP/hpx/pull/4040
https://github.com/STEllAR-GROUP/hpx/pull/4039
https://github.com/STEllAR-GROUP/hpx/pull/4038
https://github.com/STEllAR-GROUP/hpx/pull/4037
https://github.com/STEllAR-GROUP/hpx/pull/4036
https://github.com/STEllAR-GROUP/hpx/pull/4035
https://github.com/STEllAR-GROUP/hpx/pull/4028
https://github.com/STEllAR-GROUP/hpx/pull/4027
https://github.com/STEllAR-GROUP/hpx/pull/4026
https://github.com/STEllAR-GROUP/hpx/pull/4025
https://github.com/STEllAR-GROUP/hpx/pull/4024
https://github.com/STEllAR-GROUP/hpx/pull/4023
https://github.com/STEllAR-GROUP/hpx/pull/4022
https://github.com/STEllAR-GROUP/hpx/pull/4021
https://github.com/STEllAR-GROUP/hpx/pull/4020
https://github.com/STEllAR-GROUP/hpx/pull/4019
https://github.com/STEllAR-GROUP/hpx/pull/4018
https://github.com/STEllAR-GROUP/hpx/pull/4015

2.10. Releases
• PR #4014 - Future overheads
• PR #4013 - Update URL to test output conversion script
• PR #4012 - Fix CUDA compilation
• PR #4011 - Fixing cyclic dependencies between modules
• PR #4010 - Ignore stable tag on CircleCI
• PR #4009 - Check circular dependencies in a circle ci step
• PR #4008 - Extend cache aligned data to handle tuple-like data
• PR #4007 - Fixing migration for components that have actions returning a client
• PR #4006 - Move is_value_proxy.hpp to algorithms module
• PR #4004 - Shorten CTest timeout on CircleCI
• PR #4003 - Refactoring to remove (internal) dependencies
• PR #4001 - Exclude tests from all target
• PR #4000 - Module errors
• PR #3999 - Enable support for compatibility headers for logging module
• PR #3998 - Add process thread binding option
• PR #3997 - Export handle_assert function
• PR #3996 - Attempt to solve issue where -latomic does not support 128bit atomics
• PR #3993 - Make sure __LINE__ is an unsigned
• PR #3991 - Fix dependencies and flags for header tests
• PR #3990 - Documentation tags fixes
• PR #3988 - Adding missing solution folder for format module test
• PR #3987 - Move runtime-dependent functions out of command line handling
• PR #3986 - Fix CMake configuration with PAPI on

https://github.com/STEllAR-GROUP/hpx/pull/4014
https://github.com/STEllAR-GROUP/hpx/pull/4013
https://github.com/STEllAR-GROUP/hpx/pull/4012
https://github.com/STEllAR-GROUP/hpx/pull/4011
https://github.com/STEllAR-GROUP/hpx/pull/4010
https://github.com/STEllAR-GROUP/hpx/pull/4009
https://github.com/STEllAR-GROUP/hpx/pull/4008
https://github.com/STEllAR-GROUP/hpx/pull/4007
https://github.com/STEllAR-GROUP/hpx/pull/4006
https://github.com/STEllAR-GROUP/hpx/pull/4004
https://github.com/STEllAR-GROUP/hpx/pull/4003
https://github.com/STEllAR-GROUP/hpx/pull/4001
https://github.com/STEllAR-GROUP/hpx/pull/4000
https://github.com/STEllAR-GROUP/hpx/pull/3999
https://github.com/STEllAR-GROUP/hpx/pull/3998
https://github.com/STEllAR-GROUP/hpx/pull/3997
https://github.com/STEllAR-GROUP/hpx/pull/3996
https://github.com/STEllAR-GROUP/hpx/pull/3993
https://github.com/STEllAR-GROUP/hpx/pull/3991
https://github.com/STEllAR-GROUP/hpx/pull/3990
https://github.com/STEllAR-GROUP/hpx/pull/3988
https://github.com/STEllAR-GROUP/hpx/pull/3987
https://github.com/STEllAR-GROUP/hpx/pull/3986

Chapter 2. What’s so special about HPX?
• PR #3985 - Module timing
• PR #3984 - Fix default behaviour of paths in add_hpx_component
• PR #3982 - Parallel executors module
• PR #3981 - Segmented algorithms module
• PR #3980 - Module logging
• PR #3979 - Module util
• PR #3978 - Fix clang-tidy step on CircleCI
• PR #3977 - Fixing solution folders for moved components
• PR #3976 - Module format
• PR #3975 - Enable deprecation warnings on CircleCI
• PR #3974 - Fix typos in documentation
• PR #3973 - Fix compilation with GCC 9
• PR #3972 - Add condition to clone apex + use of new cmake var APEX_ROOT
• PR #3971 - Add testing module
• PR #3968 - Remove unneeded file in hardware module
• PR #3967 - Remove leftover PIC settings from main CMakeLists.txt
• PR #3966 - Add missing export option in add_hpx_module
• PR #3965 - Change current_function_helper back to non-constexpr
• PR #3964 - Fixing merge problems
• PR #3962 - Add a trait for std::array for unwrapping
• PR #3961 - Making hpx::util::tuple<Ts...> and std::tuple<Ts...> convertible
• PR #3960 - Fix compilation with CUDA 10 and GCC 6
• PR #3959 - Fix C++11 incompatibility

https://github.com/STEllAR-GROUP/hpx/pull/3985
https://github.com/STEllAR-GROUP/hpx/pull/3984
https://github.com/STEllAR-GROUP/hpx/pull/3982
https://github.com/STEllAR-GROUP/hpx/pull/3981
https://github.com/STEllAR-GROUP/hpx/pull/3980
https://github.com/STEllAR-GROUP/hpx/pull/3979
https://github.com/STEllAR-GROUP/hpx/pull/3978
https://github.com/STEllAR-GROUP/hpx/pull/3977
https://github.com/STEllAR-GROUP/hpx/pull/3976
https://github.com/STEllAR-GROUP/hpx/pull/3975
https://github.com/STEllAR-GROUP/hpx/pull/3974
https://github.com/STEllAR-GROUP/hpx/pull/3973
https://github.com/STEllAR-GROUP/hpx/pull/3972
https://github.com/STEllAR-GROUP/hpx/pull/3971
https://github.com/STEllAR-GROUP/hpx/pull/3968
https://github.com/STEllAR-GROUP/hpx/pull/3967
https://github.com/STEllAR-GROUP/hpx/pull/3966
https://github.com/STEllAR-GROUP/hpx/pull/3965
https://github.com/STEllAR-GROUP/hpx/pull/3964
https://github.com/STEllAR-GROUP/hpx/pull/3962
https://github.com/STEllAR-GROUP/hpx/pull/3961
https://github.com/STEllAR-GROUP/hpx/pull/3960
https://github.com/STEllAR-GROUP/hpx/pull/3959
HPX Documentation, master

- PR #3957\(^{1512}\) - Algorithms module
- PR #3956\(^{1513}\) - [HPX_AddModule] Fix lower name var to upper
- PR #3955\(^{1514}\) - Fix CMake configuration with examples off and tests on
- PR #3954\(^{1515}\) - Move components to separate subdirectory in root of repository
- PR #3952\(^{1516}\) - Update papi.cpp
- PR #3951\(^{1517}\) - Exclude modules header tests from all target
- PR #3950\(^{1518}\) - Adding all_reduce facility to collectives module
- PR #3949\(^{1519}\) - This adds a configuration file that will cause for stale issues to be automatically closed
- PR #3948\(^{1520}\) - Fixing ALPS environment
- PR #3947\(^{1521}\) - Add major compiler version check for building hpx as a binary package
- PR #3946\(^{1522}\) - [Modules] Move the location of the generated headers
- PR #3945\(^{1523}\) - Simplify tests and examples cmake
- PR #3943\(^{1524}\) - Remove example module
- PR #3942\(^{1525}\) - Add NOEXPORT option to add_hpx_{component,library}
- PR #3938\(^{1526}\) - Use https for CDash submissions
- PR #3937\(^{1527}\) - Add HPX_WITH_BUILD_BINARY_PACKAGE to the compiler check (refs #3935)
- PR #3936\(^{1528}\) - Fixing installation of binaries on windows
- PR #3934\(^{1529}\) - Add set function for sliding_semaphore max_difference
- PR #3933\(^{1530}\) - Remove cudadevrt from compile/link flags as it breaks downstream projects
- PR #3932\(^{1531}\) - Fixing 3929
- PR #3931\(^{1532}\) - Adding all_to_all
- PR #3930\(^{1533}\) - Add test demonstrating the use of broadcast with component actions
- PR #3928\(^{1534}\) - fixed number of tasks and number of threads for heterogeneous slurm environments

\(^{1512}\) https://github.com/STEllAR-GROUP/hpx/pull/3957
\(^{1513}\) https://github.com/STEllAR-GROUP/hpx/pull/3956
\(^{1514}\) https://github.com/STEllAR-GROUP/hpx/pull/3955
\(^{1515}\) https://github.com/STEllAR-GROUP/hpx/pull/3954
\(^{1516}\) https://github.com/STEllAR-GROUP/hpx/pull/3952
\(^{1517}\) https://github.com/STEllAR-GROUP/hpx/pull/3951
\(^{1518}\) https://github.com/STEllAR-GROUP/hpx/pull/3950
\(^{1519}\) https://github.com/STEllAR-GROUP/hpx/pull/3949
\(^{1520}\) https://github.com/STEllAR-GROUP/hpx/pull/3948
\(^{1521}\) https://github.com/STEllAR-GROUP/hpx/pull/3947
\(^{1522}\) https://github.com/STEllAR-GROUP/hpx/pull/3946
\(^{1523}\) https://github.com/STEllAR-GROUP/hpx/pull/3945
\(^{1524}\) https://github.com/STEllAR-GROUP/hpx/pull/3943
\(^{1525}\) https://github.com/STEllAR-GROUP/hpx/pull/3942
\(^{1526}\) https://github.com/STEllAR-GROUP/hpx/pull/3938
\(^{1527}\) https://github.com/STEllAR-GROUP/hpx/pull/3937
\(^{1528}\) https://github.com/STEllAR-GROUP/hpx/pull/3936
\(^{1529}\) https://github.com/STEllAR-GROUP/hpx/pull/3934
\(^{1530}\) https://github.com/STEllAR-GROUP/hpx/pull/3933
\(^{1531}\) https://github.com/STEllAR-GROUP/hpx/pull/3932
\(^{1532}\) https://github.com/STEllAR-GROUP/hpx/pull/3931
\(^{1533}\) https://github.com/STEllAR-GROUP/hpx/pull/3930
\(^{1534}\) https://github.com/STEllAR-GROUP/hpx/pull/3928

Chapter 2. What’s so special about HPX?
• PR #3927\textsuperscript{1535} - Moving Cache module’s tests into separate solution folder
• PR #3926\textsuperscript{1536} - Move unit tests to cache module
• PR #3925\textsuperscript{1537} - Move version check to config module
• PR #3924\textsuperscript{1538} - Add schedule hint executor parameters
• PR #3923\textsuperscript{1539} - Allow aligning objects bigger than the cache line size
• PR #3922\textsuperscript{1540} - Add Windows builds with Travis CI
• PR #3921\textsuperscript{1541} - Add ccls cache directory to gitignore
• PR #3920\textsuperscript{1542} - Fix \texttt{git\_external} fetching of tags
• PR #3905\textsuperscript{1543} - Correct rostambod url. Fix typo in doc
• PR #3904\textsuperscript{1544} - Fix bug in \texttt{context\_base.hpp}
• PR #3903\textsuperscript{1545} - Adding new performance counters
• PR #3902\textsuperscript{1546} - Add \texttt{add\_hpx\_module} function
• PR #3901\textsuperscript{1547} - Factoring out container remapping into a separate trait
• PR #3900\textsuperscript{1548} - Making sure errors during command line processing are properly reported and will not cause assertions
• PR #3899\textsuperscript{1549} - Remove old compatibility bases from \texttt{make\_action}
• PR #3898\textsuperscript{1550} - Make parameter size be of type \texttt{size\_t}
• PR #3897\textsuperscript{1551} - Making sure all tests are disabled if \texttt{HPX\_WITH\_TESTS=OFF}
• PR #3895\textsuperscript{1552} - Add documentation for \texttt{annotated\_function}
• PR #3894\textsuperscript{1553} - Working around VS2019 problem with \texttt{make\_action}
• PR #3892\textsuperscript{1554} - Avoid MSVC compatibility warning in internal allocator
• PR #3891\textsuperscript{1555} - Removal of the default intel config include
• PR #3888\textsuperscript{1556} - Fix \texttt{async\_customization dataflow} example and Clarify what’s being tested
• PR #3887\textsuperscript{1557} - Add Doxygen documentation

\textsuperscript{1535} \url{https://github.com/STEllAR-GROUP/hpx/pull/3927}
\textsuperscript{1536} \url{https://github.com/STEllAR-GROUP/hpx/pull/3926}
\textsuperscript{1537} \url{https://github.com/STEllAR-GROUP/hpx/pull/3925}
\textsuperscript{1538} \url{https://github.com/STEllAR-GROUP/hpx/pull/3924}
\textsuperscript{1539} \url{https://github.com/STEllAR-GROUP/hpx/pull/3923}
\textsuperscript{1540} \url{https://github.com/STEllAR-GROUP/hpx/pull/3922}
\textsuperscript{1541} \url{https://github.com/STEllAR-GROUP/hpx/pull/3921}
\textsuperscript{1542} \url{https://github.com/STEllAR-GROUP/hpx/pull/3920}
\textsuperscript{1543} \url{https://github.com/STEllAR-GROUP/hpx/pull/3905}
\textsuperscript{1544} \url{https://github.com/STEllAR-GROUP/hpx/pull/3904}
\textsuperscript{1545} \url{https://github.com/STEllAR-GROUP/hpx/pull/3903}
\textsuperscript{1546} \url{https://github.com/STEllAR-GROUP/hpx/pull/3902}
\textsuperscript{1547} \url{https://github.com/STEllAR-GROUP/hpx/pull/3901}
\textsuperscript{1548} \url{https://github.com/STEllAR-GROUP/hpx/pull/3900}
\textsuperscript{1549} \url{https://github.com/STEllAR-GROUP/hpx/pull/3899}
\textsuperscript{1550} \url{https://github.com/STEllAR-GROUP/hpx/pull/3898}
\textsuperscript{1551} \url{https://github.com/STEllAR-GROUP/hpx/pull/3897}
\textsuperscript{1552} \url{https://github.com/STEllAR-GROUP/hpx/pull/3895}
\textsuperscript{1553} \url{https://github.com/STEllAR-GROUP/hpx/pull/3894}
\textsuperscript{1554} \url{https://github.com/STEllAR-GROUP/hpx/pull/3892}
\textsuperscript{1555} \url{https://github.com/STEllAR-GROUP/hpx/pull/3891}
\textsuperscript{1556} \url{https://github.com/STEllAR-GROUP/hpx/pull/3888}
\textsuperscript{1557} \url{https://github.com/STEllAR-GROUP/hpx/pull/3887}
• PR #3882\(^{1558}\) - Minor docs fixes
• PR #3880\(^{1559}\) - Updating APEX version tag
• PR #3878\(^{1560}\) - Making sure symbols are properly exported from modules (needed for Windows/MacOS)
• PR #3877\(^{1561}\) - Documentation
• PR #3876\(^{1562}\) - Module hardware
• PR #3875\(^{1563}\) - Converted typedefs in actions submodule to using directives
• PR #3874\(^{1564}\) - Allow one to suppress target keywords in hpx\_setup\_target for backwards compatibility
• PR #3873\(^{1565}\) - Add scripts to create releases and generate lists of PRs and issues
• PR #3872\(^{1566}\) - Fix latest HTML docs location
• PR #3871\(^{1567}\) - Module cache
• PR #3869\(^{1568}\) - Post 1.3.0 version bumps
• PR #3868\(^{1569}\) - Replace the macro HPX\_ASSERT by HPX\_TEST in tests
• PR #3845\(^{1570}\) - Assertion module
• PR #3839\(^{1571}\) - Make tuple serialization non-intrusive
• PR #3832\(^{1572}\) - Config module
• PR #3799\(^{1573}\) - Remove compat namespace and its contents
• PR #3701\(^{1574}\) - MoodyCamel lockfree
• PR #3496\(^{1575}\) - Disabling MPI's (deprecated) C++ interface
• PR #3192\(^{1576}\) - Move type info into hpx\_debug namespace and add print helper functions
• PR #3159\(^{1577}\) - Support Checkpointing Components

\(^{1558}\) https://github.com/STEllAR-GROUP/hpx/pull/3882
^{1559} https://github.com/STEllAR-GROUP/hpx/pull/3880
^{1560} https://github.com/STEllAR-GROUP/hpx/pull/3878
^{1561} https://github.com/STEllAR-GROUP/hpx/pull/3877
^{1562} https://github.com/STEllAR-GROUP/hpx/pull/3876
^{1563} https://github.com/STEllAR-GROUP/hpx/pull/3875
^{1564} https://github.com/STEllAR-GROUP/hpx/pull/3874
^{1565} https://github.com/STEllAR-GROUP/hpx/pull/3873
^{1566} https://github.com/STEllAR-GROUP/hpx/pull/3872
^{1567} https://github.com/STEllAR-GROUP/hpx/pull/3870
^{1568} https://github.com/STEllAR-GROUP/hpx/pull/3869
^{1569} https://github.com/STEllAR-GROUP/hpx/pull/3868
^{1570} https://github.com/STEllAR-GROUP/hpx/pull/3845
^{1571} https://github.com/STEllAR-GROUP/hpx/pull/3839
^{1572} https://github.com/STEllAR-GROUP/hpx/pull/3832
^{1573} https://github.com/STEllAR-GROUP/hpx/pull/3799
^{1574} https://github.com/STEllAR-GROUP/hpx/pull/3701
^{1575} https://github.com/STEllAR-GROUP/hpx/pull/3496
^{1576} https://github.com/STEllAR-GROUP/hpx/pull/3192
^{1577} https://github.com/STEllAR-GROUP/hpx/pull/3159
2.10.8 *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`.
- 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 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 - shouldn’t “-faligned-new” be a usage requirement?
- Issue #3841 - Build error with msvc 19 caused by SFINAE and C++17
- Issue #3836 - master branch does not build with idle rate counters enabled
- Issue #3819 - Add debug suffix to modules built in debug mode
- Issue #3817 - `HPX_INCLUDE_DIRS` contains non-existent directory

---

1578 https://www.cmake.org
1579 https://github.com/STEllAR-GROUP/hpx/issues/3863
1580 https://github.com/STEllAR-GROUP/hpx/issues/3841
1581 https://github.com/STEllAR-GROUP/hpx/issues/3836
1582 https://github.com/STEllAR-GROUP/hpx/issues/3819
1583 https://github.com/STEllAR-GROUP/hpx/issues/3817
• Issue #3810\(^{1584}\) - Source groups are not created for files in modules
• Issue #3805\(^{1585}\) - HPX won’t compile with `-DPHX_WITH_APEX=TRUE`
• Issue #3792\(^{1586}\) - Barrier Hangs When Locality Zero not included
• Issue #3778\(^{1587}\) - Replace `throw()` with `noexcept`
• Issue #3763\(^{1588}\) - configurable sort limit per task
• Issue #3758\(^{1589}\) - dataflow doesn’t convert `future<future<T>>` to `future<T>`
• Issue #3757\(^{1590}\) - When compiling undefined reference to `hpx::hpx_check_version_1_2` HPX V1.2.1, Ubuntu 18.04.01 Server Edition
• Issue #3753\(^{1591}\) - `--hpx:list-counters=full` crashes
• Issue #3746\(^{1592}\) - Detection of MPI with pmix
• Issue #3744\(^{1593}\) - Separate spinlock from same cacheline as internal data for all LCOs
• Issue #3743\(^{1594}\) - hpxcxx’s shebang doesn’t specify the python version
• Issue #3738\(^{1595}\) - Unable to debug parcelport on a single node
• Issue #3735\(^{1596}\) - Latest master: Can’t compile in MSVC
• Issue #3731\(^{1597}\) - `util::bound` seems broken on Clang with older libstdc++
• Issue #3724\(^{1598}\) - Allow to pre-set command line options through environment
• Issue #3723\(^{1599}\) - examples/resource_partitioner build issue on master branch / ubuntu 18
• Issue #3721\(^{1600}\) - faced a building error
• Issue #3720\(^{1601}\) - Hello World example fails to link
• Issue #3719\(^{1602}\) - pkg-config produces invalid output: `-l-pthread`
• Issue #3718\(^{1603}\) - Please make the python executable configurable through cmake
• Issue #3717\(^{1604}\) - interested to contribute to the organisation
• Issue #3699\(^{1605}\) - Remove ‘HPX runtime’ executable
• Issue #3698\(^{1606}\) - Ignore all locks while handling asserts

\(^{1584}\) https://github.com/STEllAR-GROUP/hpx/issues/3810
\(^{1585}\) https://github.com/STEllAR-GROUP/hpx/issues/3805
\(^{1586}\) https://github.com/STEllAR-GROUP/hpx/issues/3792
\(^{1587}\) https://github.com/STEllAR-GROUP/hpx/issues/3778
\(^{1588}\) https://github.com/STEllAR-GROUP/hpx/issues/3763
\(^{1589}\) https://github.com/STEllAR-GROUP/hpx/issues/3758
\(^{1590}\) https://github.com/STEllAR-GROUP/hpx/issues/3757
\(^{1591}\) https://github.com/STEllAR-GROUP/hpx/issues/3753
\(^{1592}\) https://github.com/STEllAR-GROUP/hpx/issues/3746
\(^{1593}\) https://github.com/STEllAR-GROUP/hpx/issues/3744
\(^{1594}\) https://github.com/STEllAR-GROUP/hpx/issues/3743
\(^{1595}\) https://github.com/STEllAR-GROUP/hpx/issues/3738
\(^{1596}\) https://github.com/STEllAR-GROUP/hpx/issues/3735
\(^{1597}\) https://github.com/STEllAR-GROUP/hpx/issues/3731
\(^{1598}\) https://github.com/STEllAR-GROUP/hpx/issues/3724
\(^{1599}\) https://github.com/STEllAR-GROUP/hpx/issues/3723
\(^{1600}\) https://github.com/STEllAR-GROUP/hpx/issues/3721
\(^{1601}\) https://github.com/STEllAR-GROUP/hpx/issues/3720
\(^{1602}\) https://github.com/STEllAR-GROUP/hpx/issues/3719
\(^{1603}\) https://github.com/STEllAR-GROUP/hpx/issues/3718
\(^{1604}\) https://github.com/STEllAR-GROUP/hpx/issues/3717
\(^{1605}\) https://github.com/STEllAR-GROUP/hpx/issues/3699
\(^{1606}\) https://github.com/STEllAR-GROUP/hpx/issues/3698
• Issue #3689 - Incorrect and inconsistent website structure http://stellar.cct.lsu.edu/downloads/.
• Issue #3681 - Broken links on http://stellar.cct.lsu.edu/2015/05/hpx-archives-now-on-gmane/
• Issue #3676 - HPX master built from source, cmake fails to link main.cpp example in docs
• Issue #3673 - HPX build fails with std::atomic missing error
• Issue #3670 - Generate PDF again from documentation (with Sphinx)
• Issue #3643 - Warnings when compiling HPX 1.2.1 with gcc 9
• Issue #3641 - Trouble with using ranges-v3 and hpx::parallel::reduce
• Issue #3639 - util::unwrapping does not work well with member functions
• Issue #3634 - The build fails if shared_future<>::then is called with a thread executor
• Issue #3632 - VTune Amplifier 2019 not working with use_itt_notify=1
• Issue #3616 - HPX Fails to Build with CUDA 10
• Issue #3612 - False sharing of scheduling counters
• Issue #3609 - executor_parameters timeout with gcc <= 7 and Debug mode
• Issue #3601 - Misleading error message on power pc for rdtsc and rdtscp
• Issue #3598 - Build of some examples fails when using Vc
• Issue #3594 - Error: The number of OS threads requested (20) does not match the number of threads to bind (12): HPX(bad_parameter)
• Issue #3592 - Undefined Reference Error
• Issue #3589 - include could not find load file: HPX_Utils.cmake
• Issue #3587 - HPX won’t compile on POWER8 with Clang 7
• Issue #3583 - Fedora and openSUSE instructions missing on “Distribution Packages” page
• Issue #3578 - Build error when configuring with HPX_HAVE_ALGORITHM_INPUT_ITERATOR_SUPPORT=ON
• Issue #3575 - Merge openSUSE reproducible patch
• Issue #3570 - Update HPX to work with the latest VC version

https://github.com/STEllAR-GROUP/hpx/issues/3689
https://github.com/STEllAR-GROUP/hpx/issues/3681
https://github.com/STEllAR-GROUP/hpx/issues/3676
https://github.com/STEllAR-GROUP/hpx/issues/3673
https://github.com/STEllAR-GROUP/hpx/issues/3670
https://github.com/STEllAR-GROUP/hpx/issues/3643
https://github.com/STEllAR-GROUP/hpx/issues/3641
https://github.com/STEllAR-GROUP/hpx/issues/3639
https://github.com/STEllAR-GROUP/hpx/issues/3634
https://github.com/STEllAR-GROUP/hpx/issues/3622
https://github.com/STEllAR-GROUP/hpx/issues/3616
https://github.com/STEllAR-GROUP/hpx/issues/3612
https://github.com/STEllAR-GROUP/hpx/issues/3609
https://github.com/STEllAR-GROUP/hpx/issues/3601
https://github.com/STEllAR-GROUP/hpx/issues/3598
https://github.com/STEllAR-GROUP/hpx/issues/3594
https://github.com/STEllAR-GROUP/hpx/issues/3592
https://github.com/STEllAR-GROUP/hpx/issues/3589
https://github.com/STEllAR-GROUP/hpx/issues/3587
https://github.com/STEllAR-GROUP/hpx/issues/3583
https://github.com/STEllAR-GROUP/hpx/issues/3578
https://github.com/STEllAR-GROUP/hpx/issues/3575
https://github.com/STEllAR-GROUP/hpx/issues/3570
• Issue #3567: Build succeed and make failed for `hpx:cout`
• Issue #3565: Polymorphic simple component destructor not getting called
• Issue #3559: 1.2.0 is missing from download page
• Issue #3554: Clang 6.0 warning of hiding overloaded virtual function
• Issue #3510: Build on ppc64 fails
• Issue #3482: Improve error message when `HPX_WITH_MAX_CPU_COUNT` is too low for given system
• Issue #3453: Two HPX applications can’t run at the same time.
• Issue #3452: Scaling issue on the change to 2 NUMA domains
• Issue #3442: HPX `set_difference`, `set_intersection` failure cases
• Issue #3437: Ensure parent_task pointer when child task is created and child/parent are on same locality
• Issue #3255: Suspension with lock for `--hpx:list-component-types`
• Issue #3034: Use C++17 structured bindings for serialization
• Issue #2999: Change thread scheduling use of `size_t` for thread indexing

Closed pull requests

• PR #3865: adds `hpx_target_compile_option_if_available`
• PR #3864: Helper functions that are useful in numa binding and testing of allocator
• PR #3862: Temporary fix to local_dataflow_boost_small_vector test
• PR #3860: Add cache line padding to intermediate results in for loop reduction
• PR #3859: Remove `HPX_TLL_PUBLIC` and `HPX_TLL_PRIVATE` from CMake files
• PR #3858: Add compile flags and definitions to modules
• PR #3851: update hpxmp release tag to v0.2.0
• PR #3849: Correct `BOOST_ROOT` variable name in quick start guide
• PR #3847: Fix `attach_debugger` configuration option

1630 https://github.com/STEllAR-GROUP/hpx/issues/3567
1631 https://github.com/STEllAR-GROUP/hpx/issues/3565
1632 https://github.com/STEllAR-GROUP/hpx/issues/3559
1633 https://github.com/STEllAR-GROUP/hpx/issues/3554
1634 https://github.com/STEllAR-GROUP/hpx/issues/3510
1635 https://github.com/STEllAR-GROUP/hpx/issues/3482
1636 https://github.com/STEllAR-GROUP/hpx/issues/3453
1637 https://github.com/STEllAR-GROUP/hpx/issues/3452
1638 https://github.com/STEllAR-GROUP/hpx/issues/3442
1639 https://github.com/STEllAR-GROUP/hpx/issues/3437
1640 https://github.com/STEllAR-GROUP/hpx/issues/3255
1641 https://github.com/STEllAR-GROUP/hpx/issues/3034
1642 https://github.com/STEllAR-GROUP/hpx/issues/2999
1643 https://github.com/STEllAR-GROUP/hpx/pull/3865
1644 https://github.com/STEllAR-GROUP/hpx/pull/3864
1645 https://github.com/STEllAR-GROUP/hpx/pull/3862
1646 https://github.com/STEllAR-GROUP/hpx/pull/3860
1647 https://github.com/STEllAR-GROUP/hpx/pull/3859
1648 https://github.com/STEllAR-GROUP/hpx/pull/3858
1649 https://github.com/STEllAR-GROUP/hpx/pull/3851
1650 https://github.com/STEllAR-GROUP/hpx/pull/3849
1651 https://github.com/STEllAR-GROUP/hpx/pull/3847
• PR #3846\textsuperscript{1652} - Add tests for libs header tests
• PR #3844\textsuperscript{1653} - Fixing source\_groups in preprocessor module to properly handle compatibility headers
• PR #3843\textsuperscript{1654} - This fixes the launch\_process/launched\_process pair of tests
• PR #3842\textsuperscript{1655} - Fix macro call with ITTNOTIFY enabled
• PR #3840\textsuperscript{1656} - Fixing SLURM environment parsing
• PR #3837\textsuperscript{1657} - Fixing misplaced #endif
• PR #3835\textsuperscript{1658} - make all latch members protected for consistency
• PR #3834\textsuperscript{1659} - Disable transpose\_block\_numa example on CircleCI
• PR #3833\textsuperscript{1660} - make latch counter\_ protected for deriving latch in hpxmp
• PR #3831\textsuperscript{1661} - Fix CircleCI config for modules
• PR #3830\textsuperscript{1662} - minor fix: option HPX\_WITH\_TEST was not working correctly
• PR #3829\textsuperscript{1663} - Avoid for binaries that depend on HPX to directly link against internal modules
• PR #3827\textsuperscript{1664} - Adding shortcut for hpx\:\:\:get\_ptr\<\<(sync, id) for a local, non-migratable objects
• PR #3826\textsuperscript{1665} - Fix and update modules documentation
• PR #3825\textsuperscript{1666} - Updating default APEX version to 2.1.3 with HPX
• PR #3823\textsuperscript{1667} - Fix pkgconfig lib handling
• PR #3822\textsuperscript{1668} - Change includes in hpx\_wrap.cpp to more specific includes
• PR #3821\textsuperscript{1669} - Disable barrier\_3792 test when networking is disabled
• PR #3820\textsuperscript{1670} - Assorted CMake fixes
• PR #3815\textsuperscript{1671} - Removing left-over debug output
• PR #3814\textsuperscript{1672} - Allow setting default scheduler mode via the configuration database
• PR #3813\textsuperscript{1673} - Make the deprecation warnings issued by the old pp headers optional
• PR #3812\textsuperscript{1674} - Windows requires to handle symlinks to directories differently from those linking files

\textsuperscript{1652} https://github.com/STEllAR-GROUP/hpx/pull/3846
\textsuperscript{1653} https://github.com/STEllAR-GROUP/hpx/pull/3844
\textsuperscript{1654} https://github.com/STEllAR-GROUP/hpx/pull/3843
\textsuperscript{1655} https://github.com/STEllAR-GROUP/hpx/pull/3842
\textsuperscript{1656} https://github.com/STEllAR-GROUP/hpx/pull/3840
\textsuperscript{1657} https://github.com/STEllAR-GROUP/hpx/pull/3837
\textsuperscript{1658} https://github.com/STEllAR-GROUP/hpx/pull/3835
\textsuperscript{1659} https://github.com/STEllAR-GROUP/hpx/pull/3834
\textsuperscript{1660} https://github.com/STEllAR-GROUP/hpx/pull/3833
\textsuperscript{1661} https://github.com/STEllAR-GROUP/hpx/pull/3831
\textsuperscript{1662} https://github.com/STEllAR-GROUP/hpx/pull/3830
\textsuperscript{1663} https://github.com/STEllAR-GROUP/hpx/pull/3828
\textsuperscript{1664} https://github.com/STEllAR-GROUP/hpx/pull/3827
\textsuperscript{1665} https://github.com/STEllAR-GROUP/hpx/pull/3826
\textsuperscript{1666} https://github.com/STEllAR-GROUP/hpx/pull/3825
\textsuperscript{1667} https://github.com/STEllAR-GROUP/hpx/pull/3823
\textsuperscript{1668} https://github.com/STEllAR-GROUP/hpx/pull/3822
\textsuperscript{1669} https://github.com/STEllAR-GROUP/hpx/pull/3821
\textsuperscript{1670} https://github.com/STEllAR-GROUP/hpx/pull/3820
\textsuperscript{1671} https://github.com/STEllAR-GROUP/hpx/pull/3815
\textsuperscript{1672} https://github.com/STEllAR-GROUP/hpx/pull/3814
\textsuperscript{1673} https://github.com/STEllAR-GROUP/hpx/pull/3813
\textsuperscript{1674} https://github.com/STEllAR-GROUP/hpx/pull/3812
• PR #3811 - Clean up PP module and library skeleton
• PR #3806 - Moving include path configuration to before APEX
• PR #3804 - Fix latch
• PR #3803 - Update hpxcxx to look at lib64 and use python3
• PR #3802 - Numa binding allocator
• PR #3801 - Remove duplicated includes
• PR #3800 - Attempt to fix Posix context switching after lazy init changes
• PR #3798 - count and count_if accepts different iterator types
• PR #3797 - Adding a couple of override keywords to overloaded virtual functions
• PR #3796 - Re-enable testing all schedulers in shutdown_suspended_test
• PR #3795 - Change std::terminate to std::abort in SIGSEGV handler
• PR #3794 - Fixing #3792
• PR #3793 - Extending migrate_polymorphic_component unit test
• PR #3791 - Change throw() to noexcept
• PR #3790 - Remove deprecated options for 1.3.0 release
• PR #3789 - Remove Boost filesystem compatibility header
• PR #3788 - Disabled even more spots that should not execute if networking is disabled
• PR #3787 - Bump minimal boost supported version to 1.61.0
• PR #3786 - Bump minimum required versions for 1.3.0 release
• PR #3785 - Explicitly set number of jobs for all ninja invocations on CircleCI
• PR #3784 - Fix leak and address sanitizer problems
• PR #3783 - Disabled even more spots that should not execute if networking is disabled
• PR #3782 - Cherry-picked tuple and thread_init_data fixes from #3701
• PR #3781 - Fix generic context coroutines after lazy stack allocation changes
• PR #3780 - Rename hello world examples
• PR #3776 - Sort algorithms now use the supplied chunker to determine the required minimal chunk size
• PR #3775 - Disable Boost auto-linking
• PR #3774 - Tag and push stable builds
• PR #3773 - Enable migration of polymorphic components
• PR #3771 - Fix link to stackoverflow in documentation
• PR #3770 - Replacing constexpr if in brace-serialization code
• PR #3769 - Fix SIGSEGV handler
• PR #3768 - Adding flags to scheduler allowing to control thread stealing and idle back-off
• PR #3767 - Fix help formatting in hpxrun.py
• PR #3765 - Fix a couple of bugs in the thread test
• PR #3764 - Workaround for SFINAE regression in msvc14.2
• PR #3762 - Prevent MSVC from prematurely instantiating things
• PR #3761 - Update python scripts to work with python 3
• PR #3760 - Fix callable vtable for GCC4.9
• PR #3759 - Rename PAGE_SIZE to PAGE_SIZE_ because AppleClang
• PR #3755 - Making sure locks are not held during suspension
• PR #3754 - Disable more code if networking is not available/not enabled
• PR #3752 - Move util::format implementation to source file
• PR #3751 - Fixing problems with lcos::barrier and iostreams
• PR #3750 - Change error message to take into account use_guard_page setting
• PR #3749 - Fix lifetime problem in run_as_hpx_thread

https://github.com/STEllAR-GROUP/hpx/pull/3781
https://github.com/STEllAR-GROUP/hpx/pull/3780
https://github.com/STEllAR-GROUP/hpx/pull/3776
https://github.com/STEllAR-GROUP/hpx/pull/3775
https://github.com/STEllAR-GROUP/hpx/pull/3774
https://github.com/STEllAR-GROUP/hpx/pull/3773
https://github.com/STEllAR-GROUP/hpx/pull/3771
https://github.com/STEllAR-GROUP/hpx/pull/3770
https://github.com/STEllAR-GROUP/hpx/pull/3769
https://github.com/STEllAR-GROUP/hpx/pull/3768
https://github.com/STEllAR-GROUP/hpx/pull/3767
https://github.com/STEllAR-GROUP/hpx/pull/3765
https://github.com/STEllAR-GROUP/hpx/pull/3764
https://github.com/STEllAR-GROUP/hpx/pull/3762
https://github.com/STEllAR-GROUP/hpx/pull/3761
https://github.com/STEllAR-GROUP/hpx/pull/3760
https://github.com/STEllAR-GROUP/hpx/pull/3759
https://github.com/STEllAR-GROUP/hpx/pull/3755
https://github.com/STEllAR-GROUP/hpx/pull/3754
https://github.com/STEllAR-GROUP/hpx/pull/3752
https://github.com/STEllAR-GROUP/hpx/pull/3751
https://github.com/STEllAR-GROUP/hpx/pull/3750
https://github.com/STEllAR-GROUP/hpx/pull/3749
• PR #3748 - Fixed unusable behavior of the clang code analyzer.
• PR #3747 - Added PMIX_RANK to the defaults of HPX_WITH_PARCELPORT_MPI_ENV.
• PR #3745 - Introduced cache_aligned_data and cache_line_data helper structure
• PR #3742 - Remove more unused functionality from util/logging
• PR #3740 - Fix includes in partitioned vector tests
• PR #3739 - More fixes to make sure that std::flush really flushes all output
• PR #3737 - Fix potential shutdown problems
• PR #3736 - Fix guided_pool_executor after dataflow changes caused compilation fail
• PR #3734 - Limiting executor
• PR #3732 - More constrained bound constructors
• PR #3730 - Attempt to fix deadlocks during component loading
• PR #3729 - Add latch member function count_up and reset, requested by hpxMP
• PR #3728 - Send even empty buffers on hpx::endl and hpx::flush
• PR #3727 - Adding example demonstrating how to customize the memory management for a component
• PR #3726 - Adding support for passing command line options through the HPX_COMMANDLINE_OPTIONS environment variable
• PR #3722 - Document known broken OpenMPI builds
• PR #3716 - Add barrier reset function, requested by hpxMP for reusing barrier
• PR #3715 - More work on functions and vtables
• PR #3714 - Generate single-page HTML, PDF, manpage from documentation
• PR #3713 - Updating default APEX version to 2.1.2
• PR #3712 - Update release procedure
• PR #3710 - Fix the C++11 build, after #3704
• PR #3709 - Move some component_registry functionality to source file

https://github.com/STEllAR-GROUP/hpx/pull/3748
https://github.com/STEllAR-GROUP/hpx/pull/3747
https://github.com/STEllAR-GROUP/hpx/pull/3745
https://github.com/STEllAR-GROUP/hpx/pull/3742
https://github.com/STEllAR-GROUP/hpx/pull/3740
https://github.com/STEllAR-GROUP/hpx/pull/3739
https://github.com/STEllAR-GROUP/hpx/pull/3737
https://github.com/STEllAR-GROUP/hpx/pull/3736
https://github.com/STEllAR-GROUP/hpx/pull/3734
https://github.com/STEllAR-GROUP/hpx/pull/3732
https://github.com/STEllAR-GROUP/hpx/pull/3730
https://github.com/STEllAR-GROUP/hpx/pull/3729
https://github.com/STEllAR-GROUP/hpx/pull/3728
https://github.com/STEllAR-GROUP/hpx/pull/3727
https://github.com/STEllAR-GROUP/hpx/pull/3726
https://github.com/STEllAR-GROUP/hpx/pull/3722
https://github.com/STEllAR-GROUP/hpx/pull/3716
https://github.com/STEllAR-GROUP/hpx/pull/3715
https://github.com/STEllAR-GROUP/hpx/pull/3714
https://github.com/STEllAR-GROUP/hpx/pull/3713
https://github.com/STEllAR-GROUP/hpx/pull/3712
https://github.com/STEllAR-GROUP/hpx/pull/3710
https://github.com/STEllAR-GROUP/hpx/pull/3709
- PR #3708 - Ignore all locks while handling assertions
- PR #3707 - Remove obsolete hpx runtime executable
- PR #3705 - Fix and simplify make_ready_future overload sets
- PR #3704 - Reduce use of binders
- PR #3703 - Fixing CUDA compiler errors
- PR #3700 - Added barrier::increment function to increase total number of thread
- PR #3697 - One more attempt to fix migration...
- PR #3694 - Fixing component migration
- PR #3693 - Print thread state when getting disallowed value in set_thread_state
- PR #3692 - Only disable constexpr with clang-cuda, not nvcc+gcc
- PR #3691 - Link with libsupc++ if needed for thread_local
- PR #3690 - Remove thousands separators in set_operations_3442 to comply with C++11
- PR #3688 - Decouple serialization from function vtables
- PR #3687 - Fix a couple of test failures
- PR #3686 - Make sure tests.unit.build are run after install on CircleCI
- PR #3685 - Revise quickstart CMakeLists.txt explanation
- PR #3684 - Provide concept emulation for Ranges-TS concepts
- PR #3683 - Ignore uninitialized chunks
- PR #3682 - Ignore uninitialized chunks. Check proper indices.
- PR #3680 - Ignore uninitialized chunks. Check proper range indices
- PR #3679 - Simplify basic action implementations
- PR #3678 - Making sure HPX_HAVE_LIBATOMIC is unset before checking

https://github.com/STEllAR-GROUP/hpx/pull/3708
https://github.com/STEllAR-GROUP/hpx/pull/3707
https://github.com/STEllAR-GROUP/hpx/pull/3705
https://github.com/STEllAR-GROUP/hpx/pull/3704
https://github.com/STEllAR-GROUP/hpx/pull/3703
https://github.com/STEllAR-GROUP/hpx/pull/3702
https://github.com/STEllAR-GROUP/hpx/pull/3700
https://github.com/STEllAR-GROUP/hpx/pull/3697
https://github.com/STEllAR-GROUP/hpx/pull/3694
https://github.com/STEllAR-GROUP/hpx/pull/3693
https://github.com/STEllAR-GROUP/hpx/pull/3692
https://github.com/STEllAR-GROUP/hpx/pull/3691
https://github.com/STEllAR-GROUP/hpx/pull/3690
https://github.com/STEllAR-GROUP/hpx/pull/3688
https://github.com/STEllAR-GROUP/hpx/pull/3687
https://github.com/STEllAR-GROUP/hpx/pull/3686
https://github.com/STEllAR-GROUP/hpx/pull/3685
https://github.com/STEllAR-GROUP/hpx/pull/3684
https://github.com/STEllAR-GROUP/hpx/pull/3683
https://github.com/STEllAR-GROUP/hpx/pull/3682
https://github.com/STEllAR-GROUP/hpx/pull/3680
https://github.com/STEllAR-GROUP/hpx/pull/3679
https://github.com/STEllAR-GROUP/hpx/pull/3678

2.10. Releases
• PR #3677 - Fix generated full version number to be usable in expressions
• PR #3674 - Reduce functional utilities call depth
• PR #3672 - Change new build system to use existing macros related to pseudo dependencies
• PR #3669 - Remove indirection in function_ref when thread description is disabled
• PR #3668 - Unbreaking async_*cb* tests
• PR #3667 - Generate version.hpp
• PR #3665 - Enabling MPI parcelport for gitlab runners
• PR #3664 - making clang-tidy work properly again
• PR #3662 - Attempt to fix exception handling
• PR #3661 - Move lcos::latch to source file
• PR #3660 - Fix accidentally explicit gid_type default constructor
• PR #3659 - Parallel executor latch
• PR #3658 - Fixing execution_parameters
• PR #3657 - Avoid dangling references in wait_all
• PR #3656 - Avoiding lifetime problems with sync_put_parcel
• PR #3655 - Fixing nullptr dereference inside of function
• PR #3652 - Attempt to fix thread_map_type definition with C++11
• PR #3650 - Allowing for end iterator being different from begin iterator
• PR #3649 - Added architecture identification to cmake to be able to detect timestamp support
• PR #3645 - Enabling sanitizers on gitlab runner
• PR #3644 - Attempt to tackle timeouts during startup
• PR #3642 - Cleanup parallel partitioners
• PR #3640 - Dataflow now works with functions that return a reference

1767 https://github.com/STEllAR-GROUP/hpx/pull/3677
1768 https://github.com/STEllAR-GROUP/hpx/pull/3674
1769 https://github.com/STEllAR-GROUP/hpx/pull/3672
1770 https://github.com/STEllAR-GROUP/hpx/pull/3669
1771 https://github.com/STEllAR-GROUP/hpx/pull/3668
1772 https://github.com/STEllAR-GROUP/hpx/pull/3667
1773 https://github.com/STEllAR-GROUP/hpx/pull/3665
1774 https://github.com/STEllAR-GROUP/hpx/pull/3664
1775 https://github.com/STEllAR-GROUP/hpx/pull/3662
1776 https://github.com/STEllAR-GROUP/hpx/pull/3661
1777 https://github.com/STEllAR-GROUP/hpx/pull/3660
1778 https://github.com/STEllAR-GROUP/hpx/pull/3659
1779 https://github.com/STEllAR-GROUP/hpx/pull/3658
1780 https://github.com/STEllAR-GROUP/hpx/pull/3657
1781 https://github.com/STEllAR-GROUP/hpx/pull/3656
1782 https://github.com/STEllAR-GROUP/hpx/pull/3655
1783 https://github.com/STEllAR-GROUP/hpx/pull/3652
1784 https://github.com/STEllAR-GROUP/hpx/pull/3650
1785 https://github.com/STEllAR-GROUP/hpx/pull/3649
1786 https://github.com/STEllAR-GROUP/hpx/pull/3645
1787 https://github.com/STEllAR-GROUP/hpx/pull/3644
1788 https://github.com/STEllAR-GROUP/hpx/pull/3642
1789 https://github.com/STEllAR-GROUP/hpx/pull/3640
- PR #3637 - Merging the executor-enabled overloads of `shared_future<>::then`
- PR #3633 - Replace deprecated boost endian macros
- PR #3632 - Add instructions on getting HPX to documentation
- PR #3631 - Simplify parcel creation
- PR #3630 - Small additions and fixes to release procedure
- PR #3629 - Modular pp
- PR #3627 - Implement `util::function_ref`
- PR #3626 - Fix `cancelable_action_client` example
- PR #3625 - Added automatic serialization for simple structs (see #3034)
- PR #3624 - Updating the default order of priority for `thread_description`
- PR #3621 - Update copyright year and other small formatting fixes
- PR #3620 - Adding support for gitlab runner
- PR #3619 - Store debug logs and core dumps on CircleCI
- PR #3618 - Various optimizations
- PR #3617 - Fix link to the gpg key (#2)
- PR #3615 - Fix unused variable warnings with networking off
- PR #3614 - Restructuring counter data in scheduler to reduce false sharing
- PR #3613 - Adding support for gitlab runners
- PR #3610 - Don’t wait for `stop_condition` in main thread
- PR #3608 - Add `inline` keyword to `invalid_thread_id` definition for nvcc
- PR #3607 - Adding configuration key that allows one to explicitly add a directory to the component search path
- PR #3606 - Add nvcc to exclude constexpress since it is not supported by nvcc
- PR #3605 - Add `inline` to definition of checkpoint stream operators to fix link error

1790 https://github.com/STEllAR-GROUP/hpx/pull/3637
1791 https://github.com/STEllAR-GROUP/hpx/pull/3633
1792 https://github.com/STEllAR-GROUP/hpx/pull/3632
1793 https://github.com/STEllAR-GROUP/hpx/pull/3631
1794 https://github.com/STEllAR-GROUP/hpx/pull/3630
1795 https://github.com/STEllAR-GROUP/hpx/pull/3629
1796 https://github.com/STEllAR-GROUP/hpx/pull/3627
1797 https://github.com/STEllAR-GROUP/hpx/pull/3626
1798 https://github.com/STEllAR-GROUP/hpx/pull/3625
1799 https://github.com/STEllAR-GROUP/hpx/pull/3624
1800 https://github.com/STEllAR-GROUP/hpx/pull/3621
1801 https://github.com/STEllAR-GROUP/hpx/pull/3620
1802 https://github.com/STEllAR-GROUP/hpx/pull/3619
1803 https://github.com/STEllAR-GROUP/hpx/pull/3618
1804 https://github.com/STEllAR-GROUP/hpx/pull/3617
1805 https://github.com/STEllAR-GROUP/hpx/pull/3615
1806 https://github.com/STEllAR-GROUP/hpx/pull/3614
1807 https://github.com/STEllAR-GROUP/hpx/pull/3613
1808 https://github.com/STEllAR-GROUP/hpx/pull/3610
1809 https://github.com/STEllAR-GROUP/hpx/pull/3608
1810 https://github.com/STEllAR-GROUP/hpx/pull/3607
1811 https://github.com/STEllAR-GROUP/hpx/pull/3606
1812 https://github.com/STEllAR-GROUP/hpx/pull/3605

2.10. Releases
• PR #3604 - Use format for string formatting
• PR #3603 - Improve the error message for using to less MAX_CPU_COUNT
• PR #3602 - Improve the error message for to small values of MAX_CPU_COUNT
• PR #3600 - Parallel executor aggregated
• PR #3599 - Making sure networking is disabled for default one-locality-runs
• PR #3596 - Store thread exit functions in forward_list instead of deque to avoid allocations
• PR #3590 - Fix typo/mistake in thread queue cleanup_terminated
• PR #3588 - Fix formatting errors in launching_and_configuring_hpx_applications.rst
• PR #3586 - Make bind propagate value category
• PR #3585 - Extend Cmake for building hpx as distribution packages (refs #3575)
• PR #3584 - Untangle function storage from object pointer
• PR #3582 - Towards Modularized HPX
• PR #3580 - Remove extra || in merge.hpp
• PR #3577 - Partially revert “Remove vtable empty flag”
• PR #3576 - Make sure empty startup/shutdown functions are not being used
• PR #3574 - Make sure DATAPAR settings are conveyed to depending projects
• PR #3573 - Make sure HPX is usable with latest released version of Vc (V1.4.1)
• PR #3572 - Adding test ensuring ticket 3565 is fixed
• PR #3571 - Make empty [unique_]function vtable non-dependent
• PR #3566 - Fix compilation with dynamic bitset for CPU masks
• PR #3563 - Drop util::[unique_]function target_type
• PR #3562 - Removing the target suffixes
• PR #3561 - Replace executor traits return type deduction (keep non-SFINAE)

1813 https://github.com/STEllAR-GROUP/hpx/pull/3604
1814 https://github.com/STEllAR-GROUP/hpx/pull/3603
1815 https://github.com/STEllAR-GROUP/hpx/pull/3602
1816 https://github.com/STEllAR-GROUP/hpx/pull/3600
1817 https://github.com/STEllAR-GROUP/hpx/pull/3599
1818 https://github.com/STEllAR-GROUP/hpx/pull/3596
1819 https://github.com/STEllAR-GROUP/hpx/pull/3590
1820 https://github.com/STEllAR-GROUP/hpx/pull/3588
1821 https://github.com/STEllAR-GROUP/hpx/pull/3586
1822 https://github.com/STEllAR-GROUP/hpx/pull/3585
1823 https://github.com/STEllAR-GROUP/hpx/pull/3584
1824 https://github.com/STEllAR-GROUP/hpx/pull/3582
1825 https://github.com/STEllAR-GROUP/hpx/pull/3580
1826 https://github.com/STEllAR-GROUP/hpx/pull/3577
1827 https://github.com/STEllAR-GROUP/hpx/pull/3576
1828 https://github.com/STEllAR-GROUP/hpx/pull/3574
1829 https://github.com/STEllAR-GROUP/hpx/pull/3573
1830 https://github.com/STEllAR-GROUP/hpx/pull/3572
1831 https://github.com/STEllAR-GROUP/hpx/pull/3571
1832 https://github.com/STEllAR-GROUP/hpx/pull/3566
1833 https://github.com/STEllAR-GROUP/hpx/pull/3563
1834 https://github.com/STEllAR-GROUP/hpx/pull/3562
1835 https://github.com/STEllAR-GROUP/hpx/pull/3561
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.

2.10.9 HPX V1.2.1 (Feb 19, 2019)
• Add support for Boost 1.69.0.

Closed issues

• Issue #3638 - Build HPX 1.2 with boost 1.69
• Issue #3635 - Non-deterministic crashing on Stampede2
• Issue #3550 - 1>e:000\work\hpx\src\throw_exception.cpp(54): error C2440: ‘<function-style-cast>’: cannot convert from ‘boost::system::error_code’ to ‘hpx::exception’
• Issue #3549 - HPX 1.2.0 does not build on i686, but release candidate did
• Issue #3511 - Build on s390x fails
• Issue #3509 - Build on armv7l fails

Closed pull requests

• PR #3695 - Don’t install CMake templates and packaging files
• PR #3666 - Fixing yet another race in future_data
• PR #3663 - Fixing race between setting and getting the value inside future_data
• PR #3648 - Adding timestamp option for S390x platform
• PR #3647 - Blind attempt to fix warnings issued by gcc V9
• PR #3611 - Include GNUInstallDirs earlier to have it available for subdirectories
• PR #3595 - Use GNUInstallDirs lib path in pkgconfig config file
• PR #3593 - Add include(GNUInstallDirs) to HPX Macros cmake
• PR #3591 - Fix compilation error on arm7 architecture. Compiles and runs on Fedora 29 on Pi 3.
• PR #3558 - Adding constructor exception(boost::system::error_code const&)
• PR #3555 - cmake: make install locations configurable
• PR #3551 - Fix uint64_t causing compilation fail on i686
2.10.10 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\(^{1872}\) or pkg-config. The old behaviour can be restored by setting `HPX_WITH_DYNAMIC_HPX_MAIN=OFF` during CMake\(^{1873}\) configuration. The implementation on Windows is unchanged.

• 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\(^{1874}\) 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\(^{1875}\) 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\(^{1876}\) 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.

\(^{1872}\) https://www.cmake.org
\(^{1873}\) https://www.cmake.org
\(^{1874}\) https://www.cmake.org
\(^{1875}\) https://www.cmake.org
\(^{1876}\) https://www.cmake.org
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\(^{1877}\).

- 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`.

Closed issues

- Issue #3538\(^{1878}\) - numa handling incorrect for hwloc 2
- Issue #3533\(^{1879}\) - Cmake version 3.5.1 does not work (git ff26b35 2018-11-06)
- Issue #3526\(^{1880}\) - Failed building hpx-1.2.0-rc1 on Ubuntu16.04 x86-64 Virtualbox VM
- Issue #3512\(^{1881}\) - Build on aarch64 fails
- Issue #3475\(^{1882}\) - HPX fails to link if the MPI parcelport is enabled
- Issue #3462\(^{1883}\) - CMake configuration shows a minor and inconsequential failure to create a symlink
- Issue #3461\(^{1884}\) - Compilation Problems with the most recent Clang
- Issue #3460\(^{1885}\) - Deadlock when create_partitioner fails (assertion fails) in debug mode
- Issue #3455\(^{1886}\) - HPX build failing with HWLOC errors on POWER8 with hwloc 1.8
- Issue #3438\(^{1887}\) - HPX no longer builds on IBM POWER8
- Issue #3426\(^{1888}\) - hpx build failed on MacOS
- Issue #3424\(^{1889}\) - CircleCI builds broken for forked repositories
- Issue #3422\(^{1890}\) - Benchmarks in tests.performance.local are not run nightly

\(^{1877}\) https://github.com/STEllAR-GROUP/hpx/pull/3266

\(^{1878}\) https://github.com/STEllAR-GROUP/hpx/issues/3538

\(^{1879}\) https://github.com/STEllAR-GROUP/hpx/issues/3533

\(^{1880}\) https://github.com/STEllAR-GROUP/hpx/issues/3526

\(^{1881}\) https://github.com/STEllAR-GROUP/hpx/issues/3512

\(^{1882}\) https://github.com/STEllAR-GROUP/hpx/issues/3475

\(^{1883}\) https://github.com/STEllAR-GROUP/hpx/issues/3462

\(^{1884}\) https://github.com/STEllAR-GROUP/hpx/issues/3461

\(^{1885}\) https://github.com/STEllAR-GROUP/hpx/issues/3460

\(^{1886}\) https://github.com/STEllAR-GROUP/hpx/issues/3455

\(^{1887}\) https://github.com/STEllAR-GROUP/hpx/issues/3438

\(^{1888}\) https://github.com/STEllAR-GROUP/hpx/issues/3426

\(^{1889}\) https://github.com/STEllAR-GROUP/hpx/issues/3424

\(^{1890}\) https://github.com/STEllAR-GROUP/hpx/issues/3422
• Issue #3408 - CMake Targets for HPX
• Issue #3399 - processing unit out of bounds
• Issue #3395 - Floating point bug in hpx/runtime/threads/policies/scheduler_base.hpp
• Issue #3378 - compile error with lcos::communicator
• Issue #3376 - Failed to build HPX with APEX using clang
• Issue #3366 - Adapted Safe_Object example fails for --hpx:threads > 1
• Issue #3360 - Segmentation fault when passing component id as parameter
• Issue #3358 - HPX runtime hangs after multiple (~thousands) start-stop sequences
• Issue #3352 - Support TCP provider in libfabric ParcelPort
• Issue #3342 - undefined reference to __atomic_load_16
• Issue #3339 - setting command line options/flags from init cfg is not obvious
• Issue #3325 - AGAS migrates components prematurely
• Issue #3321 - hpx bad parameter handling is awful
• Issue #3318 - Benchmarks fail to build with C++11
• Issue #3304 - hpx::threads::run_as_hpx_thread does not properly handle exceptions
• Issue #3300 - Setting pu step or offset results in no threads in default pool
• Issue #3297 - Crash with APEX when running Phylanx lra_csv with > 1 thread
• Issue #3296 - Building HPX with APEX configuration gives compiler warnings
• Issue #3290 - make tests failing at hello_world_component
• Issue #3285 - possible compilation error when “using namespace std;” is defined before including “hpx”
• Issue #3280 - HPX fails on OSX
• Issue #3272 - CircleCI does not upload generated docker image any more
• Issue #3270 - Error when compiling CUDA examples
• Issue #3267\textsuperscript{1914} - \texttt{tests.unit.host_.block_allocator} fails occasionally
• Issue #3264\textsuperscript{1915} - Possible move to Sphinx for documentation
• Issue #3263\textsuperscript{1916} - Documentation improvements
• Issue #3259\textsuperscript{1917} - \texttt{setParcelWriteHandler} test fails occasionally
• Issue #3258\textsuperscript{1918} - Links to source code in documentation are broken
• Issue #3247\textsuperscript{1919} - Rare \texttt{tests.unit.host_.block_allocator} test failure on 1.1.0-rc1
• Issue #3244\textsuperscript{1920} - Slowing down and speeding up an interval\_timer
• Issue #3215\textsuperscript{1921} - Cannot build both tests and examples on MSVC with pseudo-dependencies enabled
• Issue #3195\textsuperscript{1922} - Unnecessary customization point route causing performance penalty
• Issue #3088\textsuperscript{1923} - A strange thing in parallel::sort.
• Issue #2650\textsuperscript{1924} - libfabric support for passive endpoints
• Issue #1205\textsuperscript{1925} - TSS is broken

Closed pull requests

• PR #3542\textsuperscript{1926} - Fix numa lookup from pu when using hwloc 2.x
• PR #3541\textsuperscript{1927} - Fixing the build system of the MPI parcelport
• PR #3540\textsuperscript{1928} - Updating HPX people section
• PR #3539\textsuperscript{1929} - Splitting test to avoid OOM on CircleCI
• PR #3537\textsuperscript{1930} - Fix guided exec
• PR #3536\textsuperscript{1931} - Updating grants which support the LSU team
• PR #3535\textsuperscript{1932} - Fix hiding of docker credentials
• PR #3534\textsuperscript{1933} - Fixing #3533
• PR #3532\textsuperscript{1934} - fixing minor doc typo --hpx:print-counter-at arg
• PR #3530\textsuperscript{1935} - Changing APEX default tag to v2.1.0

\textsuperscript{1914}https://github.com/STEllAR-GROUP/hpx/issues/3267
\textsuperscript{1915}https://github.com/STEllAR-GROUP/hpx/issues/3264
\textsuperscript{1916}https://github.com/STEllAR-GROUP/hpx/issues/3263
\textsuperscript{1917}https://github.com/STEllAR-GROUP/hpx/issues/3259
\textsuperscript{1918}https://github.com/STEllAR-GROUP/hpx/issues/3258
\textsuperscript{1919}https://github.com/STEllAR-GROUP/hpx/issues/3247
\textsuperscript{1920}https://github.com/STEllAR-GROUP/hpx/issues/3244
\textsuperscript{1921}https://github.com/STEllAR-GROUP/hpx/issues/3215
\textsuperscript{1922}https://github.com/STEllAR-GROUP/hpx/issues/3195
\textsuperscript{1923}https://github.com/STEllAR-GROUP/hpx/issues/3088
\textsuperscript{1924}https://github.com/STEllAR-GROUP/hpx/issues/2650
\textsuperscript{1925}https://github.com/STEllAR-GROUP/hpx/issues/1205
\textsuperscript{1926}https://github.com/STEllAR-GROUP/hpx/pull/3542
\textsuperscript{1927}https://github.com/STEllAR-GROUP/hpx/pull/3541
\textsuperscript{1928}https://github.com/STEllAR-GROUP/hpx/pull/3540
\textsuperscript{1929}https://github.com/STEllAR-GROUP/hpx/pull/3539
\textsuperscript{1930}https://github.com/STEllAR-GROUP/hpx/pull/3537
\textsuperscript{1931}https://github.com/STEllAR-GROUP/hpx/pull/3536
\textsuperscript{1932}https://github.com/STEllAR-GROUP/hpx/pull/3535
\textsuperscript{1933}https://github.com/STEllAR-GROUP/hpx/pull/3534
\textsuperscript{1934}https://github.com/STEllAR-GROUP/hpx/pull/3532
\textsuperscript{1935}https://github.com/STEllAR-GROUP/hpx/pull/3530

Chapter 2. What’s so special about \textit{HPX}?
• PR #3529 - Remove leftover security options and documentation
• PR #3528 - Fix hwloc version check
• PR #3524 - Do not build guided pool examples with older GCC compilers
• PR #3523 - Fix logging regression
• PR #3522 - Fix more warnings
• PR #3521 - Fixing argument handling in induction and reduction clauses for parallel::for_loop
• PR #3520 - Remove docs symlink and versioned docs folders
• PR #3519 - hpxMP release
• PR #3518 - Change all steps to use new docker image on CircleCI
• PR #3516 - Drop usage of deprecated facilities removed in C++17
• PR #3515 - Remove remaining uses of Boost.TypeTraits
• PR #3513 - Fixing a CMake problem when trying to use libfabric
• PR #3508 - Remove memory_block component
• PR #3507 - Propagating the MPI compile definitions to all relevant targets
• PR #3503 - Update documentation colors and logo
• PR #3502 - Fix bogus `throws` bindings in scheduled_thread_pool_impl
• PR #3501 - Split parallel::remove_if tests to avoid OOM on CircleCI
• PR #3500 - Support NONAMEPREFIX in add_hpx_library()
• PR #3497 - Note that cuda support requires cmake 3.9
• PR #3495 - Fixing dataflow
• PR #3493 - Remove deprecated options for 1.2.0 part 2
• PR #3492 - Add CUDA_LINK_LIBRARIES_KEYWOR to allow PRIVATE keyword in linkage t...
• PR #3491 - Changing Base docker image

https://github.com/STEllAR-GROUP/hpx/pull/3529
https://github.com/STEllAR-GROUP/hpx/pull/3528
https://github.com/STEllAR-GROUP/hpx/pull/3524
https://github.com/STEllAR-GROUP/hpx/pull/3523
https://github.com/STEllAR-GROUP/hpx/pull/3522
https://github.com/STEllAR-GROUP/hpx/pull/3521
https://github.com/STEllAR-GROUP/hpx/pull/3520
https://github.com/STEllAR-GROUP/hpx/pull/3519
https://github.com/STEllAR-GROUP/hpx/pull/3518
https://github.com/STEllAR-GROUP/hpx/pull/3516
https://github.com/STEllAR-GROUP/hpx/pull/3515
https://github.com/STEllAR-GROUP/hpx/pull/3513
https://github.com/STEllAR-GROUP/hpx/pull/3508
https://github.com/STEllAR-GROUP/hpx/pull/3507
https://github.com/STEllAR-GROUP/hpx/pull/3503
https://github.com/STEllAR-GROUP/hpx/pull/3502
https://github.com/STEllAR-GROUP/hpx/pull/3501
https://github.com/STEllAR-GROUP/hpx/pull/3500
https://github.com/STEllAR-GROUP/hpx/pull/3497
https://github.com/STEllAR-GROUP/hpx/pull/3495
https://github.com/STEllAR-GROUP/hpx/pull/3493
https://github.com/STEllAR-GROUP/hpx/pull/3492
https://github.com/STEllAR-GROUP/hpx/pull/3491

2.10. Releases
• PR #3490\(^{1959}\) - Don’t create tasks immediately with hpx::apply
• PR #3489\(^{1960}\) - Remove deprecated options for 1.2.0
• PR #3488\(^{1961}\) - Revert “Use BUILD_INTERFACE generator expression to fix cmake flag exports”
• PR #3487\(^{1962}\) - Revert “Fixing type attribute warning for transfer_action”
• PR #3485\(^{1963}\) - Use BUILD_INTERFACE generator expression to fix cmake flag exports
• PR #3483\(^{1964}\) - Fixing type attribute warning for transfer_action
• PR #3481\(^{1965}\) - Remove unused variables
• PR #3480\(^{1966}\) - Towards a more lightweight transfer action
• PR #3479\(^{1967}\) - Fix FLAGS - Use correct version of target_compile_options
• PR #3478\(^{1968}\) - Making sure the application’s exit code is properly propagated back to the OS
• PR #3476\(^{1969}\) - Don’t print docker credentials as part of the environment.
• PR #3473\(^{1970}\) - Fixing invalid cmake code if no jemalloc prefix was given
• PR #3472\(^{1971}\) - Attempting to work around recent clang test compilation failures
• PR #3471\(^{1972}\) - Enable jemalloc on windows
• PR #3470\(^{1973}\) - Updates readme
• PR #3468\(^{1974}\) - Avoid hang if there is an exception thrown during startup
• PR #3467\(^{1975}\) - Add compiler specific fallthrough attributes if C++17 attribute is not available
• PR #3466\(^{1976}\) - - bugfix : fix compilation with llvm-7.0
• PR #3465\(^{1977}\) - This patch adds various optimizations extracted from the thread_local_allocator work
• PR #3464\(^{1978}\) - Check for forked repos in CircleCI docker push step
• PR #3463\(^{1979}\) - - cmake : create the parent directory before symlinking
• PR #3459\(^{1980}\) - Remove unused/incomplete functionality from util/logging
• PR #3458\(^{1981}\) - Fix a problem with scope of CMAKE_CXX_FLAGS and hpx_add_compile_flag

\(^{1959}\) https://github.com/STEllAR-GROUP/hpx/pull/3490
\(^{1960}\) https://github.com/STEllAR-GROUP/hpx/pull/3489
\(^{1961}\) https://github.com/STEllAR-GROUP/hpx/pull/3488
\(^{1962}\) https://github.com/STEllAR-GROUP/hpx/pull/3487
\(^{1963}\) https://github.com/STEllAR-GROUP/hpx/pull/3485
\(^{1964}\) https://github.com/STEllAR-GROUP/hpx/pull/3483
\(^{1965}\) https://github.com/STEllAR-GROUP/hpx/pull/3481
\(^{1966}\) https://github.com/STEllAR-GROUP/hpx/pull/3480
\(^{1967}\) https://github.com/STEllAR-GROUP/hpx/pull/3479
\(^{1968}\) https://github.com/STEllAR-GROUP/hpx/pull/3478
\(^{1969}\) https://github.com/STEllAR-GROUP/hpx/pull/3476
\(^{1970}\) https://github.com/STEllAR-GROUP/hpx/pull/3473
\(^{1971}\) https://github.com/STEllAR-GROUP/hpx/pull/3472
\(^{1972}\) https://github.com/STEllAR-GROUP/hpx/pull/3471
\(^{1973}\) https://github.com/STEllAR-GROUP/hpx/pull/3470
\(^{1974}\) https://github.com/STEllAR-GROUP/hpx/pull/3468
\(^{1975}\) https://github.com/STEllAR-GROUP/hpx/pull/3467
\(^{1976}\) https://github.com/STEllAR-GROUP/hpx/pull/3466
\(^{1977}\) https://github.com/STEllAR-GROUP/hpx/pull/3465
\(^{1978}\) https://github.com/STEllAR-GROUP/hpx/pull/3464
\(^{1979}\) https://github.com/STEllAR-GROUP/hpx/pull/3463
\(^{1980}\) https://github.com/STEllAR-GROUP/hpx/pull/3459
\(^{1981}\) https://github.com/STEllAR-GROUP/hpx/pull/3458
• PR #3457\textsuperscript{1982} - Fixing more size_t -> int16_t (and similar) warnings
• PR #3456\textsuperscript{1983} - Add #ifdefs to topology.cpp to support old hwloc versions again
• PR #3454\textsuperscript{1984} - Fixing warnings related to silent conversion of size_t -> int16_t
• PR #3451\textsuperscript{1985} - Add examples as unit tests
• PR #3450\textsuperscript{1986} - Constexpr-fying bind and other functional facilities
• PR #3446\textsuperscript{1987} - Fix some thread suspension timeouts
• PR #3445\textsuperscript{1988} - Fix various warnings
• PR #3443\textsuperscript{1989} - Only enable service pool config options if pools are enabled
• PR #3441\textsuperscript{1990} - Fix missing closing brackets in documentation
• PR #3439\textsuperscript{1991} - Use correct MPI CXX libraries for MPI parcelport
• PR #3436\textsuperscript{1992} - Add projection function to find_* (and fix very bad bug)
• PR #3435\textsuperscript{1993} - Fixing 1205
• PR #3434\textsuperscript{1994} - Fix threads cores
• PR #3433\textsuperscript{1995} - Add Heise Online to release announcement list
• PR #3432\textsuperscript{1996} - Don’t track task dependencies for distributed runs
• PR #3431\textsuperscript{1997} - Circle CI setting changes for hpxMP
• PR #3430\textsuperscript{1998} - Fix unused params warning
• PR #3429\textsuperscript{1999} - One thread per core
• PR #3428\textsuperscript{2000} - This suppresses a deprecation warning that is being issued by MSVC 19.15.26726
• PR #3427\textsuperscript{2001} - Fixes #3426
• PR #3425\textsuperscript{2002} - Use source cache and workspace between job steps on CircleCI
• PR #3421\textsuperscript{2003} - Add CDash timing output to future overhead test (for graphs)
• PR #3420\textsuperscript{2004} - Add guided\_pool\_executor

\textsuperscript{1982} https://github.com/STEllAR-GROUP/hpx/pull/3457
\textsuperscript{1983} https://github.com/STEllAR-GROUP/hpx/pull/3456
\textsuperscript{1984} https://github.com/STEllAR-GROUP/hpx/pull/3454
\textsuperscript{1985} https://github.com/STEllAR-GROUP/hpx/pull/3451
\textsuperscript{1986} https://github.com/STEllAR-GROUP/hpx/pull/3450
\textsuperscript{1987} https://github.com/STEllAR-GROUP/hpx/pull/3446
\textsuperscript{1988} https://github.com/STEllAR-GROUP/hpx/pull/3445
\textsuperscript{1989} https://github.com/STEllAR-GROUP/hpx/pull/3443
\textsuperscript{1990} https://github.com/STEllAR-GROUP/hpx/pull/3441
\textsuperscript{1991} https://github.com/STEllAR-GROUP/hpx/pull/3439
\textsuperscript{1992} https://github.com/STEllAR-GROUP/hpx/pull/3436
\textsuperscript{1993} https://github.com/STEllAR-GROUP/hpx/pull/3435
\textsuperscript{1994} https://github.com/STEllAR-GROUP/hpx/pull/3434
\textsuperscript{1995} https://github.com/STEllAR-GROUP/hpx/pull/3433
\textsuperscript{1996} https://github.com/STEllAR-GROUP/hpx/pull/3432
\textsuperscript{1997} https://github.com/STEllAR-GROUP/hpx/pull/3431
\textsuperscript{1998} https://github.com/STEllAR-GROUP/hpx/pull/3430
\textsuperscript{1999} https://github.com/STEllAR-GROUP/hpx/pull/3429
\textsuperscript{2000} https://github.com/STEllAR-GROUP/hpx/pull/3428
\textsuperscript{2001} https://github.com/STEllAR-GROUP/hpx/pull/3427
\textsuperscript{2002} https://github.com/STEllAR-GROUP/hpx/pull/3425
\textsuperscript{2003} https://github.com/STEllAR-GROUP/hpx/pull/3421
\textsuperscript{2004} https://github.com/STEllAR-GROUP/hpx/pull/3420

2.10. Releases
• PR #3419<sup>2005</sup> - Fix typo in CircleCI config
• PR #3418<sup>2006</sup> - Add sphinx documentation
• PR #3415<sup>2007</sup> - Scheduler NUMA hint and shared priority scheduler
• PR #3414<sup>2008</sup> - Adding step to synchronize the APEX release
• PR #3413<sup>2009</sup> - Fixing multiple defines of APEX_HAVE_HPX
• PR #3412<sup>2010</sup> - Fixes linking with libhpx_wrap error with BSD and Windows based systems
• PR #3410<sup>2011</sup> - Fix typo in CMakeLists.txt
• PR #3409<sup>2012</sup> - Fix brackets and indentation in existing_performance_counters.qbk
• PR #3407<sup>2013</sup> - Fix unused param and extra ; warnings emitted by gcc 8.x
• PR #3406<sup>2014</sup> - Adding thread local allocator and use it for future shared states
• PR #3405<sup>2015</sup> - Adding DHPX_HAVE_THREAD_LOCAL_STORAGE=ON to builds
• PR #3404<sup>2016</sup> - fixing multiple definition of main() in linux
• PR #3402<sup>2017</sup> - Allow debug option to be enabled only for Linux systems with dynamic main on
• PR #3401<sup>2018</sup> - Fix cuda_future_helper.h when compiling with C++11
• PR #3400<sup>2019</sup> - Fix floating point exception scheduler_base idle backoff
• PR #3398<sup>2020</sup> - Atomic future state
• PR #3397<sup>2021</sup> - Fixing code for older gcc versions
• PR #3396<sup>2022</sup> - Allowing to register thread event functions (start/stop/error)
• PR #3394<sup>2023</sup> - Fix small mistake in primary_namespace_server.cpp
• PR #3393<sup>2024</sup> - Explicitly instantiate configured schedulers
• PR #3392<sup>2025</sup> - Add performance counters background overhead and background work duration
• PR #3391<sup>2026</sup> - Adapt integration of HPXMP to latest build system changes
• PR #3390<sup>2027</sup> - Make AGAS measurements optional

<sup>2005</sup> https://github.com/STEllAR-GROUP/hpx/pull/3419
<sup>2006</sup> https://github.com/STEllAR-GROUP/hpx/pull/3418
<sup>2007</sup> https://github.com/STEllAR-GROUP/hpx/pull/3415
<sup>2008</sup> https://github.com/STEllAR-GROUP/hpx/pull/3414
<sup>2009</sup> https://github.com/STEllAR-GROUP/hpx/pull/3413
<sup>2010</sup> https://github.com/STEllAR-GROUP/hpx/pull/3412
<sup>2011</sup> https://github.com/STEllAR-GROUP/hpx/pull/3410
<sup>2012</sup> https://github.com/STEllAR-GROUP/hpx/pull/3409
<sup>2013</sup> https://github.com/STEllAR-GROUP/hpx/pull/3407
<sup>2014</sup> https://github.com/STEllAR-GROUP/hpx/pull/3406
<sup>2015</sup> https://github.com/STEllAR-GROUP/hpx/pull/3405
<sup>2016</sup> https://github.com/STEllAR-GROUP/hpx/pull/3404
<sup>2017</sup> https://github.com/STEllAR-GROUP/hpx/pull/3402
<sup>2018</sup> https://github.com/STEllAR-GROUP/hpx/pull/3401
<sup>2019</sup> https://github.com/STEllAR-GROUP/hpx/pull/3400
<sup>2020</sup> https://github.com/STEllAR-GROUP/hpx/pull/3398
<sup>2021</sup> https://github.com/STEllAR-GROUP/hpx/pull/3397
<sup>2022</sup> https://github.com/STEllAR-GROUP/hpx/pull/3396
<sup>2023</sup> https://github.com/STEllAR-GROUP/hpx/pull/3394
<sup>2024</sup> https://github.com/STEllAR-GROUP/hpx/pull/3393
<sup>2025</sup> https://github.com/STEllAR-GROUP/hpx/pull/3392
<sup>2026</sup> https://github.com/STEllAR-GROUP/hpx/pull/3391
<sup>2027</sup> https://github.com/STEllAR-GROUP/hpx/pull/3390
- Fix deadlock during shutdown
- Add several functionalities allowing to optimize synchronous action invocation
- Add cmake option to opt out of fail-compile tests
- Adding support for boost::container::small_vector to dataflow
- Adds Debug option for hpx initializing from main
- This hopefully fixes two tests that occasionally fail
- Making sure thread local storage is enable for hpxMP
- Fix usage of HPX_CAPTURE together with default value capture [=]
- Replace undefined instantiations of uniform_int_distribution
- Add missing semicolons to uses of HPX_COMPILER_FENCE
- Fixing #3378
- Adding build system support to integrate hpxmp into hpx at the user’s machine
- Replacing wrapper for __libc_start_main with main
- Adds hpx_wrap to HPX_LINK_LIBRARY which links only when specified.
- Forcing cache settings in HPXConfig.cmake to guarantee updated values
- Fix some more c++11 build problems
- Adds HPX_LINKER_FLAGS to HPX applications without editing their source codes
- util::format: add type_specifier<> specializations for %!s(MISSING) and %!l(MISSING)s
- Adding configuration option to allow explicit disable of the new hpx_main feature on Linux
- Updates doc with recent hpx_wrap implementation
- Adds Mac OS implementation to hpx_main.hpp
- Fix order of hpx libs in HPX_CONF_LIBRARY.
- Apex fixing null wrapper

2.10. Releases

https://github.com/STEllAR-GROUP/hpx/pull/3389
https://github.com/STEllAR-GROUP/hpx/pull/3388
https://github.com/STEllAR-GROUP/hpx/pull/3387
https://github.com/STEllAR-GROUP/hpx/pull/3386
https://github.com/STEllAR-GROUP/hpx/pull/3385
https://github.com/STEllAR-GROUP/hpx/pull/3384
https://github.com/STEllAR-GROUP/hpx/pull/3383
https://github.com/STEllAR-GROUP/hpx/pull/3382
https://github.com/STEllAR-GROUP/hpx/pull/3381
https://github.com/STEllAR-GROUP/hpx/pull/3380
https://github.com/STEllAR-GROUP/hpx/pull/3379
https://github.com/STEllAR-GROUP/hpx/pull/3377
https://github.com/STEllAR-GROUP/hpx/pull/3376
https://github.com/STEllAR-GROUP/hpx/pull/3375
https://github.com/STEllAR-GROUP/hpx/pull/3374
https://github.com/STEllAR-GROUP/hpx/pull/3373
https://github.com/STEllAR-GROUP/hpx/pull/3372
https://github.com/STEllAR-GROUP/hpx/pull/3371
https://github.com/STEllAR-GROUP/hpx/pull/3370
https://github.com/STEllAR-GROUP/hpx/pull/3369
https://github.com/STEllAR-GROUP/hpx/pull/3368
https://github.com/STEllAR-GROUP/hpx/pull/3367
https://github.com/STEllAR-GROUP/hpx/pull/3366
https://github.com/STEllAR-GROUP/hpx/pull/3365
https://github.com/STEllAR-GROUP/hpx/pull/3364
https://github.com/STEllAR-GROUP/hpx/pull/3363
• PR #3361 - Making sure all parcels get destroyed on an HPX thread (TCP pp)
• PR #3359 - Feature/improveerrorforcompiler
• PR #3357 - Static/dynamic executable implementation
• PR #3355 - Reverting changes introduced by #3283 as those make applications hang
• PR #3354 - Add external dependencies to HPX_LIBRARY_DIR
• PR #3353 - Fix libfabric tcp
• PR #3351 - Move obsolete header to tests directory.
• PR #3350 - Renaming two functions to avoid problem described in #3285
• PR #3349 - Make idle backoff exponential with maximum sleep time
• PR #3347 - Replace simple_component* with component* in the Documentation
• PR #3346 - Fix CMakeLists.txt example in quick start
• PR #3345 - Fix automatic setting of HPX_MORE_THAN_64_THREADS
• PR #3344 - Reduce amount of information printed for unknown command line options
• PR #3343 - Safeguard HPX against destruction in global contexts
• PR #3341 - Allowing for all command line options to be used as configuration settings
• PR #3340 - Always convert inspect results to JUnit XML
• PR #3336 - Only run docker push on master on CircleCI
• PR #3335 - Update description of hpx.os_threads config parameter.
• PR #3334 - Making sure early logging settings don’t get mixed with others
• PR #3333 - Update CMake links and versions in documentation
• PR #3332 - Add notes on target suffixes to CMake documentation
• PR #3331 - Add quickstart section to documentation
• PR #3330 - Rename resource_partitioner test to avoid conflicts with pseudodependencies

https://github.com/STEllAR-GROUP/hpx/pull/3361
https://github.com/STEllAR-GROUP/hpx/pull/3359
https://github.com/STEllAR-GROUP/hpx/pull/3357
https://github.com/STEllAR-GROUP/hpx/pull/3355
https://github.com/STEllAR-GROUP/hpx/pull/3354
https://github.com/STEllAR-GROUP/hpx/pull/3353
https://github.com/STEllAR-GROUP/hpx/pull/3351
https://github.com/STEllAR-GROUP/hpx/pull/3350
https://github.com/STEllAR-GROUP/hpx/pull/3349
https://github.com/STEllAR-GROUP/hpx/pull/3347
https://github.com/STEllAR-GROUP/hpx/pull/3346
https://github.com/STEllAR-GROUP/hpx/pull/3345
https://github.com/STEllAR-GROUP/hpx/pull/3344
https://github.com/STEllAR-GROUP/hpx/pull/3343
https://github.com/STEllAR-GROUP/hpx/pull/3341
https://github.com/STEllAR-GROUP/hpx/pull/3340
https://github.com/STEllAR-GROUP/hpx/pull/3336
https://github.com/STEllAR-GROUP/hpx/pull/3335
https://github.com/STEllAR-GROUP/hpx/pull/3334
https://github.com/STEllAR-GROUP/hpx/pull/3333
https://github.com/STEllAR-GROUP/hpx/pull/3332
https://github.com/STEllAR-GROUP/hpx/pull/3331
https://github.com/STEllAR-GROUP/hpx/pull/3330
- PR #3328 - Making sure object is pinned while executing actions, even if action returns a future
- PR #3327 - Add missing std::forward to tuple.hpp
- PR #3326 - Make sure logging is up and running while modules are being discovered.
- PR #3324 - Replace C++14 overload of std::equal with C++11 code.
- PR #3323 - Fix a missing apex thread data (wrapper) initialization
- PR #3320 - Adding support for -std=c++2a (define `HPX_WITH_CXX2A=On`)
- PR #3319 - Replacing C++14 feature with equivalent C++11 code
- PR #3317 - Fix compilation with VS 15.7.1 and /std:c++latest
- PR #3316 - Fix includes for 1d_stencil_*_omp examples
- PR #3314 - Remove some unused parameter warnings
- PR #3313 - Fix pu-step and pu-offset command line options
- PR #3312 - Add conversion of inspect reports to JUnit XML
- PR #3311 - Fix escaping of closing braces in format specification syntax
- PR #3310 - Don’t overwrite user settings with defaults in registration database
- PR #3309 - Fixing potential stack overflow for dataflow
- PR #3308 - This updates the .clang-format configuration file to utilize newer features
- PR #3306 - Marking migratable objects in their gid to allow not handling migration in AGAS
- PR #3305 - Add proper exception handling to run_as_hpx_thread
- PR #3303 - Changed std::rand to a better inbuilt PRNG Generator
- PR #3302 - All non-migratable (simple) components now encode their lva and component type in their gid
- PR #3301 - Add nullptr_t overloads to resource partitioner
- PR #3298 - Apex task wrapper memory bug
- PR #3295 - Fix mistakes after merge of CircleCI config
• PR #3294 - Fix partitioned vector include in partitioned_vector_find tests
• PR #3293 - Adding emplace support to promise and make_ready_future
• PR #3292 - Add new cuda kernel synchronization with hpx::future demo
• PR #3291 - Fixes #3290
• PR #3280 - Fixing Docker image creation
• PR #3288 - Avoid allocating shared state for wait_all
• PR #3287 - Fixing /scheduler/utilization/instantaneous performance counter
• PR #3286 - dataflow() and future::then() use sync policy where possible
• PR #3284 - Background thread can use relaxed atomics to manipulate thread state
• PR #3283 - Do not unwrap ready future
• PR #3282 - Fix virtual method override warnings in static schedulers
• PR #3281 - Disable set_area_membind_nodeset for OSX
• PR #3279 - Add two variations to the future_overhead benchmark
• PR #3278 - Fix circleci workspace
• PR #3277 - Support external plugins
• PR #3276 - Fix missing parenthesis in hello_compute.cu.
• PR #3274 - Reinit counters synchronously in reinit_counters test
• PR #3273 - Splitting tests to avoid compiler OOM
• PR #3271 - Remove leftover code from context_generic_context.hpp
• PR #3269 - Fix bulk_construct with count = 0
• PR #3268 - Replace constexpr with HPX_CXX14_CONSTEXPR and HPX_CONSTEXPR
• PR #3266 - Replace boost::format with custom sprintf-based implementation
• PR #3265 - Split parallel tests on CircleCI

2097 https://github.com/STEllAR-GROUP/hpx/pull/3294
2098 https://github.com/STEllAR-GROUP/hpx/pull/3293
2099 https://github.com/STEllAR-GROUP/hpx/pull/3292
2100 https://github.com/STEllAR-GROUP/hpx/pull/3291
2101 https://github.com/STEllAR-GROUP/hpx/pull/3289
2102 https://github.com/STEllAR-GROUP/hpx/pull/3288
2103 https://github.com/STEllAR-GROUP/hpx/pull/3287
2104 https://github.com/STEllAR-GROUP/hpx/pull/3286
2105 https://github.com/STEllAR-GROUP/hpx/pull/3284
2106 https://github.com/STEllAR-GROUP/hpx/pull/3283
2107 https://github.com/STEllAR-GROUP/hpx/pull/3282
2108 https://github.com/STEllAR-GROUP/hpx/pull/3281
2109 https://github.com/STEllAR-GROUP/hpx/pull/3279
2110 https://github.com/STEllAR-GROUP/hpx/pull/3278
2111 https://github.com/STEllAR-GROUP/hpx/pull/3277
2112 https://github.com/STEllAR-GROUP/hpx/pull/3276
2113 https://github.com/STEllAR-GROUP/hpx/pull/3274
2114 https://github.com/STEllAR-GROUP/hpx/pull/3273
2115 https://github.com/STEllAR-GROUP/hpx/pull/3271
2116 https://github.com/STEllAR-GROUP/hpx/pull/3269
2117 https://github.com/STEllAR-GROUP/hpx/pull/3268
2118 https://github.com/STEllAR-GROUP/hpx/pull/3266
2119 https://github.com/STEllAR-GROUP/hpx/pull/3265
• PR #3262 - Making sure documentation correctly links to source files
• PR #3261 - Apex refactoring fix rebind
• PR #3260 - Isolate performance counter parser into a separate TU
• PR #3256 - Post 1.1.0 version bumps
• PR #3254 - Adding trait for actions allowing to make runtime decision on whether to execute it directly
• PR #3253 - Bump minimal supported Boost to 1.58.0
• PR #3251 - Adds new feature: changing interval used in interval_timer (issue 3244)
• PR #3239 - Changing std::rand() to a better inbuilt PRNG generator.
• PR #3234 - Disable background thread when networking is off
• PR #3232 - Clean up suspension tests
• PR #3230 - Add optional scheduler mode parameter to create_thread_pool function
• PR #3228 - Allow suspension also on static schedulers
• PR #3163 - libfabric parcelport w/o HPX_PARCELPORT_LIBFABRIC_ENDPOINT_RDM
• PR #3036 - Switching to CircleCI 2.0

2.10.11 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 and/or MPI.

• 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. We have also continued to implement these algorithms for the distributed use case (for segmented data structures, such as `hpx::partitioned_vector`).

https://openmp.org/wp/
https://en.wikipedia.org/wiki/Message_Passing_Interface
http://www.open-std.org/jtc1/sc22/wg21
- 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** 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**.

- 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**.

- 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** 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 one to 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** 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`.  

---

2137 https://www.cmake.org
2138 https://www.cmake.org
2139 https://www.cmake.org
2140 https://www.cmake.org

1688 Chapter 2. What’s so special about **HPX**?
• 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<hpx::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.

### Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #3250\(^{2141}\) - Apex refactoring with guids
- PR #3249\(^{2142}\) - Updating People.qbk
- PR #3246\(^{2143}\) - Assorted fixes for CUDA
- PR #3245\(^{2144}\) - Apex refactoring with guids
- PR #3242\(^{2145}\) - Modify task counting in thread_queue.hpp
- PR #3240\(^{2146}\) - Fixed typos
- PR #3238\(^{2147}\) - Readding accidentally removed std::abort
- PR #3237\(^{2148}\) - Adding Pipeline example
- PR #3236\(^{2149}\) - Fixing memory_block
- PR #3233\(^{2150}\) - Make schedule_thread take suspended threads into account
- Issue #3226\(^{2151}\) - memory_block is breaking, signaling SIGSEGV on a thread on creation and freeing
- PR #3225\(^{2152}\) - Applying quick fix for hwloc-2.0
- Issue #3224\(^{2153}\) - HPX counters crashing the application
- PR #3223\(^{2154}\) - Fix returns when setting config entries
- Issue #3222\(^{2155}\) - Errors linking libhpx.so

\(^{2141}\) https://github.com/STEllAR-GROUP/hpx/pull/3250
\(^{2142}\) https://github.com/STEllAR-GROUP/hpx/pull/3249
\(^{2143}\) https://github.com/STEllAR-GROUP/hpx/pull/3246
\(^{2144}\) https://github.com/STEllAR-GROUP/hpx/pull/3245
\(^{2145}\) https://github.com/STEllAR-GROUP/hpx/pull/3242
\(^{2146}\) https://github.com/STEllAR-GROUP/hpx/pull/3240
\(^{2147}\) https://github.com/STEllAR-GROUP/hpx/pull/3238
\(^{2148}\) https://github.com/STEllAR-GROUP/hpx/pull/3237
\(^{2149}\) https://github.com/STEllAR-GROUP/hpx/pull/3236
\(^{2150}\) https://github.com/STEllAR-GROUP/hpx/pull/3233
\(^{2151}\) https://github.com/STEllAR-GROUP/hpx/issues/3226
\(^{2152}\) https://github.com/STEllAR-GROUP/hpx/pull/3225
\(^{2153}\) https://github.com/STEllAR-GROUP/hpx/issues/3224
\(^{2154}\) https://github.com/STEllAR-GROUP/hpx/pull/3223
\(^{2155}\) https://github.com/STEllAR-GROUP/hpx/issues/3222
• Issue #3221 - HPX on Mac OS X with HWLoc 2.0.0 fails to run
• PR #3216 - Reorder a variadic array to satisfy VS 2017 15.6
• PR #3214 - Changed prerequisites.qbk to avoid confusion while building boost
• PR #3213 - Relax locks for thread suspension to avoid holding locks when yielding
• PR #3212 - Fix check in sequenced_executor test
• PR #3211 - Use preinit_array to set argc/argv in init_globally example
• PR #3210 - Adapted parallel::{search | search_n} for Ranges TS (see #1668)
• PR #3209 - Fix locking problems during shutdown
• Issue #3208 - init_globally throwing a run-time error
• PR #3206 - Addition of new arithmetic performance counter “Count”
• PR #3205 - Fixing return type calculation for bulk_then_execute
• PR #3204 - Changing std::rand() to a better inbuilt PRNG generator
• PR #3203 - Resolving problems during shutdown for VS2015
• PR #3202 - Making sure resource partitioner is not accessed if its not valid
• PR #3201 - Fixing optional::swap
• Issue #3200 - hpx::util::optional fails
• PR #3199 - Fix sliding_semaphore test
• PR #3198 - Set pre_main status before launching run_helper
• PR #3197 - Update README.rst
• PR #3194 - parallel::{fill|fill_n} updated for Ranges TS
• PR #3193 - Updating Runtime.cpp by adding correct description of Performance counters during register
• PR #3191 - Fix sliding_semaphore_2338 test
• PR #3190 - Topology improvements
- PR #3189 - Deleting one include of median from BOOST library to arithmetics_counter file
- PR #3188 - Optionally disable printing of diagnostics during terminate
- PR #3187 - Supressing cmake warning issued by cmake > V3.11
- PR #3185 - Remove unused scoped_unlock, unlock_guard_try
- PR #3184 - Fix nqueen example
- PR #3183 - Add runtime start/stop, resume/suspend and OpenMP benchmarks
- Issue #3182 - bulk_then_execute has unexpected return type/does not compile
- Issue #3181 - hwloc 2.0 breaks topo class and cannot be used
- Issue #3180 - Schedulers that don’t support suspend/resume are unusable
- PR #3179 - Various minor changes to support FLeCSI
- PR #3178 - Fix #3124
- PR #3177 - Removed allgather
- PR #3176 - Fixed Documentation for “using_hpx_pkgconfig”
- PR #3174 - Add hpx::iostreams::ostream overload to format_to
- PR #3172 - Fix lifo queue backend
- PR #3171 - adding the missing unset() function to cpu_mask() for case of more than 64 threads
- PR #3170 - Add cmake flag -DHPX_WITH_FAULT_TOLERANCE=ON (OFF by default)
- PR #3169 - Adapted parallel::[count|count_if] for Ranges TS (see #1668)
- PR #3168 - Changing used namespace for seq execution policy
- Issue #3167 - Update GSoc projects
- Issue #3166 - Application (Octotiger) gets stuck on hpx::finalize when only using one thread
- Issue #3165 - Compilation of parallel algorithms with HPX_WITH_DATAPAR is broken
- PR #3164 - Fixing component migration

2179 https://github.com/STEllAR-GROUP/hpx/pull/3189
2180 https://github.com/STEllAR-GROUP/hpx/pull/3188
2181 https://github.com/STEllAR-GROUP/hpx/pull/3187
2182 https://github.com/STEllAR-GROUP/hpx/pull/3185
2183 https://github.com/STEllAR-GROUP/hpx/pull/3184
2184 https://github.com/STEllAR-GROUP/hpx/pull/3183
2185 https://github.com/STEllAR-GROUP/hpx/issues/3182
2186 https://github.com/STEllAR-GROUP/hpx/issues/3181
2187 https://github.com/STEllAR-GROUP/hpx/issues/3180
2188 https://github.com/STEllAR-GROUP/hpx/pull/3179
2189 https://github.com/STEllAR-GROUP/hpx/pull/3178
2190 https://github.com/STEllAR-GROUP/hpx/pull/3177
2191 https://github.com/STEllAR-GROUP/hpx/pull/3176
2192 https://github.com/STEllAR-GROUP/hpx/pull/3174
2193 https://github.com/STEllAR-GROUP/hpx/pull/3172
2194 https://github.com/STEllAR-GROUP/hpx/pull/3171
2195 https://github.com/STEllAR-GROUP/hpx/pull/3170
2196 https://github.com/STEllAR-GROUP/hpx/pull/3169
2197 https://github.com/STEllAR-GROUP/hpx/pull/3168
2198 https://github.com/STEllAR-GROUP/hpx/issues/3167
2199 https://github.com/STEllAR-GROUP/hpx/issues/3166
2200 https://github.com/STEllAR-GROUP/hpx/issues/3165
2201 https://github.com/STEllAR-GROUP/hpx/pull/3164
• PR #3162 - regex_from_pattern: escape regex special characters to avoid misinterpretation
• Issue #3161 - Building HPX with hwloc 2.0.0 fails
• PR #3160 - Fixing the handling of quoted command line arguments.
• PR #3158 - Fixing a race with timed suspension (second attempt)
• PR #3157 - Revert “Fixing a race with timed suspension”
• PR #3156 - Fixing serialization of classes with incompatible serialize signature
• PR #3154 - More refactorings based on clang-tidy reports
• PR #3153 - Fixing a race with timed suspension
• PR #3152 - Documentation for runtime suspension
• PR #3151 - Use small_vector only from boost version 1.59 onwards
• PR #3150 - Avoiding more stack overflows
• PR #3149 - Refactoring component_base and base_action/transfer_base_action
• PR #3148 - Move yield_while out of detail namespace and into own file
• PR #3147 - Remove a leftover of the cxx11 std array cleanup
• PR #3146 - Minor changes to how actions are executed
• PR #3143 - Fix stack overhead
• PR #3142 - Fix typo in config.hpp
• PR #3141 - Fixing small_vector compatibility with older boost version
• PR #3140 - is_heap_text fix
• Issue #3139 - Error in is_heap_tests.hpp
• PR #3138 - Partially reverting #3126
• PR #3137 - Suspend speedup
• PR #3136 - Revert “Fixing #2325”
• PR #3135\(^{2225}\) - Improving destruction of threads
• Issue #3134\(^{2226}\) - HPX_SERIALIZATION_SPLIT_FREE does not stop compiler from looking for serialize() method
• PR #3133\(^{2227}\) - Make hwloc compulsory
• PR #3132\(^{2228}\) - Update CXX14 constexpr feature test
• PR #3131\(^{2229}\) - Fixing #2325
• PR #3130\(^{2230}\) - Avoid completion handler allocation
• PR #3129\(^{2231}\) - Suspend runtime
• PR #3128\(^{2232}\) - Make docbook dtd and xsl path names consistent
• PR #3127\(^{2233}\) - Add hpx::start nullptr overloads
• PR #3126\(^{2234}\) - Cleaning up coroutine implementation
• PR #3125\(^{2235}\) - Replacing nullptr with hpx::threads::invalid_thread_id
• Issue #3124\(^{2236}\) - Add hello_world_component to CI builds
• PR #3123\(^{2237}\) - Add new constructor.
• PR #3122\(^{2238}\) - Fixing #3121
• Issue #3121\(^{2239}\) - HPX_SMT_PAUSE is broken on non-x86 platforms when __GNUC__ is defined
• PR #3120\(^{2240}\) - Don’t use boost::intrusive_ptr for thread_id_type
• PR #3119\(^{2241}\) - Disable default executor compatibility with V1 executors
• PR #3118\(^{2242}\) - Adding performance_counter::reinit to allow for dynamically changing counter sets
• PR #3117\(^{2243}\) - Replace uses of boost/experimental::optional with util::optional
• PR #3116\(^{2244}\) - Moving background thread APEX timer #2980
• PR #3115\(^{2245}\) - Fixing race condition in channel test
• PR #3114\(^{2246}\) - Avoid using util::function for thread function wrappers
• PR #3113\(^{2247}\) - cmake V3.10.2 has changed the variable names used for MPI

\(^{2225}\) https://github.com/STEllAR-GROUP/hpx/pull/3135
\(^{2226}\) https://github.com/STEllAR-GROUP/hpx/issues/3134
\(^{2227}\) https://github.com/STEllAR-GROUP/hpx/pull/3133
\(^{2228}\) https://github.com/STEllAR-GROUP/hpx/pull/3132
\(^{2229}\) https://github.com/STEllAR-GROUP/hpx/pull/3131
\(^{2230}\) https://github.com/STEllAR-GROUP/hpx/pull/3130
\(^{2231}\) https://github.com/STEllAR-GROUP/hpx/pull/3129
\(^{2232}\) https://github.com/STEllAR-GROUP/hpx/pull/3128
\(^{2233}\) https://github.com/STEllAR-GROUP/hpx/pull/3127
\(^{2234}\) https://github.com/STEllAR-GROUP/hpx/pull/3126
\(^{2235}\) https://github.com/STEllAR-GROUP/hpx/pull/3125
\(^{2236}\) https://github.com/STEllAR-GROUP/hpx/issues/3124
\(^{2237}\) https://github.com/STEllAR-GROUP/hpx/pull/3123
\(^{2238}\) https://github.com/STEllAR-GROUP/hpx/pull/3122
\(^{2239}\) https://github.com/STEllAR-GROUP/hpx/issues/3121
\(^{2240}\) https://github.com/STEllAR-GROUP/hpx/pull/3120
\(^{2241}\) https://github.com/STEllAR-GROUP/hpx/pull/3119
\(^{2242}\) https://github.com/STEllAR-GROUP/hpx/pull/3118
\(^{2243}\) https://github.com/STEllAR-GROUP/hpx/pull/3117
\(^{2244}\) https://github.com/STEllAR-GROUP/hpx/pull/3116
\(^{2245}\) https://github.com/STEllAR-GROUP/hpx/pull/3115
\(^{2246}\) https://github.com/STEllAR-GROUP/hpx/pull/3114
\(^{2247}\) https://github.com/STEllAR-GROUP/hpx/pull/3113

2.10. Releases
• PR #3112 2248 - Minor fixes to exclusive_scan algorithm
• PR #3111 2249 - Revert “fix detection of cxx11_std_atomic”
• PR #3110 2250 - Suspend thread pool
• PR #3109 2251 - Fixing thread scheduling when yielding a thread id
• PR #3108 2252 - Revert “Suspend thread pool”
• PR #3107 2253 - Remove UB from thread::id relational operators
• PR #3106 2254 - Add cmake test for std::decay_t to fix cuda build
• PR #3105 2255 - Fixing refcount for async traversal frame
• PR #3104 2256 - Local execution of direct actions is now actually performed directly
• PR #3103 2257 - Adding support for generic counter_raw_values performance counter type
• Issue #3102 2258 - Introduce generic performance counter type returning an array of values
• PR #3101 2259 - Revert “Adapting stack overhead limit for gcc 4.9”
• PR #3100 2260 - Fix #3068 (condition_variable deadlock)
• PR #3099 2261 - Fixing lock held during suspension in papi counter component
• PR #3098 2262 - Unbreak broadcast_wait_for_2822 test
• PR #3097 2263 - Adapting stack overhead limit for gcc 4.9
• PR #3096 2264 - fix detection of cxx11_std_atomic
• PR #3095 2265 - Add ciso646 header to get _LIBCPP_VERSION for testing inplace merge
• PR #3094 2266 - Relax atomic operations on performance counter values
• PR #3093 2267 - Short-circuit all_of/any_of/none_of instantiations
• PR #3092 2268 - Take advantage of C++14 lambda capture initialization syntax, where possible
• PR #3091 2269 - Remove more references to Boost from logging code
• PR #3090 2270 - Unify use of yield/yield_k

2248 https://github.com/STEllAR-GROUP/hpx/pull/3112
2249 https://github.com/STEllAR-GROUP/hpx/pull/3111
2250 https://github.com/STEllAR-GROUP/hpx/pull/3110
2251 https://github.com/STEllAR-GROUP/hpx/pull/3109
2252 https://github.com/STEllAR-GROUP/hpx/pull/3108
2253 https://github.com/STEllAR-GROUP/hpx/pull/3107
2254 https://github.com/STEllAR-GROUP/hpx/pull/3106
2255 https://github.com/STEllAR-GROUP/hpx/pull/3105
2256 https://github.com/STEllAR-GROUP/hpx/pull/3104
2257 https://github.com/STEllAR-GROUP/hpx/pull/3103
2258 https://github.com/STEllAR-GROUP/hpx/issues/3102
2259 https://github.com/STEllAR-GROUP/hpx/pull/3101
2260 https://github.com/STEllAR-GROUP/hpx/pull/3100
2261 https://github.com/STEllAR-GROUP/hpx/pull/3099
2262 https://github.com/STEllAR-GROUP/hpx/pull/3098
2263 https://github.com/STEllAR-GROUP/hpx/pull/3097
2264 https://github.com/STEllAR-GROUP/hpx/pull/3096
2265 https://github.com/STEllAR-GROUP/hpx/pull/3095
2266 https://github.com/STEllAR-GROUP/hpx/pull/3094
2267 https://github.com/STEllAR-GROUP/hpx/pull/3093
2268 https://github.com/STEllAR-GROUP/hpx/pull/3092
2269 https://github.com/STEllAR-GROUP/hpx/pull/3091
2270 https://github.com/STEllAR-GROUP/hpx/pull/3090

1694 Chapter 2. What’s so special about HPX?
• PR #3089\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3089} - Fix a strange thing in parallel::detail::handle_exception. (Fix #2834.)
• Issue #3088\footnote{https://github.com/STEllAR-GROUP/hpx/issues/3088} - A strange thing in parallel::sort.
• PR #3087\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3087} - Fixing assertion in default_distribution_policy
• PR #3086\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3086} - Implement parallel::remove and parallel::remove_if
• PR #3085\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3085} - Addressing breaking changes in Boost V1.66
• PR #3084\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3084} - Ignore build warnings round 2
• PR #3083\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3083} - Fix typo HPX_WITH_MM_PREFECH
• PR #3081\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3081} - Pre-decay template arguments early
• PR #3080\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3080} - Suspend thread pool
• PR #3079\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3079} - Ignore build warnings
• PR #3078\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3078} - Don’t test inplace_merge with libc++
• PR #3076\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3076} - Fixing 3075: Part 1
• PR #3074\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3074} - Fix more build warnings
• PR #3073\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3073} - Suspend thread cleanup
• PR #3072\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3072} - Change existing symbol_namespace::iterate to return all data instead of invoking a callback
• PR #3071\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3071} - Fixing pack_traversal_async test
• PR #3070\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3070} - Fix dynamic_counters_loaded_1508 test by adding dependency to memory_component
• PR #3069\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3069} - Fix scheduling loop exit
• Issue #3068\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3068} - hpx::lcos::condition_variable could be suspect to deadlocks
• PR #3067\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3067} - #ifdef out random_shuffle deprecated in later c++
• PR #3066\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3066} - Make coalescing test depend on coalescing library to ensure it gets built
• PR #3065\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3065} - Workaround for minimal_timed_async_executor_test compilation failures, attempts to copy a deferred call (in unevaluated context)
• PR #3064\footnote{https://github.com/STEllAR-GROUP/hpx/pull/3064} - Fixing wrong condition in wrapper_heap
• PR #3062 - Fix exception handling for execution::seq
• PR #3061 - Adapt MSVC C++ mode handling to VS15.5
• PR #3060 - Fix compiler problem in MSVC release mode
• PR #3059 - Fixing #2931
• Issue #3058 - minimal_timed_async_executor_test_exe fails to compile on master (d6f505c)
• PR #3057 - Fix stable_merge_2964 compilation problems
• PR #3056 - Fix some build warnings caused by unused variables/unnecessary tests
• PR #3055 - Update documentation for running tests
• Issue #3054 - Assertion failure when using bulk hpx::new_ in asynchronous mode
• PR #3053 - Do not bind test running to cmake test build rule
• PR #3051 - Fix HPX-Qt interaction in Qt example.
• Issue #3048 - nqueen example fails occasionally
• PR #3047 - Fixing #3044
• PR #3046 - Add OS thread suspension
• PR #3042 - PyCicle - first attempt at a build toold for checking PR’s
• PR #3041 - Fix a problem about asynchronous execution of parallel::merge and parallel::partition.
• PR #3040 - Fix a mistake about exception handling in asynchronous execution of scan_partitioner.
• PR #3039 - Consistently use executors to schedule work
• PR #3038 - Fixing local direct function execution and lambda actions perfect forwarding
• PR #3035 - Make parallel unit test names match build target/folder names
• PR #3032 - Fix setting of default build type
• Issue #3031 - Fix partitioner arg copy found in #2982
• Issue #3031 - Errors linking libhpx.so due to missing references (master branch, commit 6679a8882)
• PR #3030⁵⁴ - Revert “implement executor then interface with && forwarding reference”
• PR #3029⁵⁴ - Run CI inspect checks before building
• PR #3028⁵³ - Added range version of parallel::move
• Issue #3027⁵⁶ - Implement all scheduling APIs in terms of executors
• PR #3026⁵⁴ - implement executor then interface with && forwarding reference
• PR #3025⁵⁴ - Fix typo uninitialized to uninitialized
• PR #3024⁵³ - Inspect fixes
• PR #3023⁵³ - P0356 Simplified partial function application
• PR #3022⁵³ - Master fixes
• PR #3021⁵³ - Segfault fix
• PR #3020⁵³ - Disable command-line aliasing for applications that use user_main
• PR #3019⁵³ - Adding enable_elasticity option to pool configuration
• PR #3018⁵³ - Fix stack overflow detection configuration in header files
• PR #3017⁵³ - Speed up local action execution
• PR #3016⁵³ - Unify stack-overflow detection options, remove reference to libsigsegv
• PR #3015⁵³ - Speeding up accessing the resource partitioner and the topology info
• Issue #3014⁵⁴ - HPX does not compile on POWER8 with gcc 5.4
• Issue #3013⁵⁴ - hello_world occasionally prints multiple lines from a single OS-thread
• PR #3012⁵⁴ - Silence warning about casting away qualifiers in itt_notify.hpp
• PR #3011⁵⁴ - Fix cpuset leak in hwloc_topology_info.cpp
• PR #3010⁵⁴ - Remove useless decay_copy
• PR #3009⁵⁴ - Fixing 2996
• PR #3008⁵⁴ - Remove unused internal function

2.10. Releases
• PR #3007 - Fixing wrapper_heap alignment problems
• Issue #3006 - hwloc memory leak
• PR #3004 - Silence C4251 (needs to have dll-interface) for future_data_void
• Issue #3003 - Suspension of runtime
• PR #3001 - Attempting to avoid data races in async_traversal while evaluating dataflow()
• PR #3000 - Adding hpx::util::optional as a first step to replace experimental::optional
• PR #2998 - Cleanup up and Fixing component creation and deletion
• Issue #2996 - Build fails with HPX_WITH_HWLOC=OFF
• PR #2995 - Push more future_data functionality to source file
• PR #2994 - WIP: Fix throttle test
• PR #2993 - Making sure –hpx:help does not throw for required (but missing) arguments
• PR #2992 - Adding non-blocking (on destruction) service executors
• Issue #2991 - run_as_os_thread locks up
• Issue #2990 - –help will not work until all required options are provided
• PR #2989 - Improve error messages caused by misuse of dataflow
• PR #2988 - Improve error messages caused by misuse of .then
• Issue #2987 - stack overflow detection producing false positives
• PR #2986 - Deduplicate non-dependent thread_info logging types
• PR #2985 - Adapted parallel::{all_of|any_of|none_of} for Ranges TS (see #1668)
• PR #2984 - Refactor one_size_heap code to simplify code
• PR #2983 - Fixing local_new_component
• PR #2982 - Clang tidy
• PR #2981 - Simplify allocator rebinding in pack traversal
- PR #2979 - Fixing integer overflows
- PR #2978 - Implement parallel::inplace_merge
- Issue #2977 - Make hwloc compulsory instead of optional
- PR #2976 - Making sure client_base instance that registered the component does not unregister it when being destructed
- PR #2975 - Change version of pulled APEX to master
- PR #2974 - Fix domain not being freed at the end of scheduling loop
- PR #2973 - Fix small typos
- PR #2972 - Adding uintstd.h header
- PR #2971 - Fall back to creating local components using local_new
- PR #2970 - Improve is_tuple_like trait
- PR #2969 - Fix HPX_WITH_MORE_THAN_64_THREADS default value
- PR #2968 - Cleaning up dataflow overload set
- PR #2967 - Make parallel::merge is stable. (Fix #2964.)
- PR #2966 - Fixing a couple of held locks during exception handling
- PR #2965 - Adding missing #include
- Issue #2964 - parallel merge is not stable
- PR #2963 - Making sure any function object passed to dataflow is released after being invoked
- PR #2962 - Partially reverting #2891
- PR #2961 - Attempt to fix the gcc 4.9 problem with the async pack traversal
- Issue #2959 - Program terminates during error handling
- Issue #2958 - HPX_PLAIN_ACTION breaks due to missing include
- PR #2957 - Fixing errors generated by mixing different attribute syntaxes
- Issue #2956 - Mixing attribute syntaxes leads to compiler errors
• Issue #2955 - Fix OS-Thread throttling
• PR #2953 - Making sure any hpx.os_threads=N supplied through a --hpx::config file is taken into account
• PR #2952 - Removing wrong call to cleanup_terminated_locked
• PR #2951 - Revert “Make sure the function vtables are initialized before use”
• PR #2950 - Fix a namespace compilation error when some schedulers are disabled
• Issue #2949 - master branch giving lockups on shutdown
• Issue #2947 - hpx.ini is not used correctly at initialization
• PR #2946 - Adding explicit feature test for thread_local
• PR #2945 - Make sure the function vtables are initialized before use
• PR #2944 - Attempting to solve affinity problems on CircleCI
• PR #2943 - Changing channel actions to be direct
• PR #2942 - Adding split_future for std::vector
• PR #2941 - Add a feature test to test for CXX11 override
• Issue #2940 - Add split_future for future<vector<T>>
• PR #2939 - Making error reporting during problems with setting affinity masks more verbose
• PR #2938 - Fix this various executors
• PR #2937 - Fix some typos in documentation
• PR #2936 - Remove the need for “complete” SFINAE checks
• PR #2935 - Making sure parallel::for_loop is executed in parallel if requested
• PR #2934 - Classify chunk_size_iterator to input iterator tag. (Fix #2866)
• Issue #2931 - --hpx:help triggers unusual error with clang build
• PR #2930 - Add #include files needed to set _POSIX_VERSION for debug check
• PR #2929 - Fix a couple of deprecated c++ features

2386 https://github.com/STEllAR-GROUP/hpx/issues/2955
2387 https://github.com/STEllAR-GROUP/hpx/pull/2953
2388 https://github.com/STEllAR-GROUP/hpx/pull/2952
2389 https://github.com/STEllAR-GROUP/hpx/pull/2951
2390 https://github.com/STEllAR-GROUP/hpx/pull/2950
2391 https://github.com/STEllAR-GROUP/hpx/issues/2949
2392 https://github.com/STEllAR-GROUP/hpx/pull/2947
2393 https://github.com/STEllAR-GROUP/hpx/pull/2946
2394 https://github.com/STEllAR-GROUP/hpx/pull/2945
2395 https://github.com/STEllAR-GROUP/hpx/pull/2944
2396 https://github.com/STEllAR-GROUP/hpx/pull/2943
2397 https://github.com/STEllAR-GROUP/hpx/pull/2942
2398 https://github.com/STEllAR-GROUP/hpx/pull/2941
2399 https://github.com/STEllAR-GROUP/hpx/issues/2940
2400 https://github.com/STEllAR-GROUP/hpx/pull/2939
2401 https://github.com/STEllAR-GROUP/hpx/pull/2938
2402 https://github.com/STEllAR-GROUP/hpx/pull/2937
2403 https://github.com/STEllAR-GROUP/hpx/pull/2934
2404 https://github.com/STEllAR-GROUP/hpx/pull/2933
2405 https://github.com/STEllAR-GROUP/hpx/pull/2932
2406 https://github.com/STEllAR-GROUP/hpx/issues/2931
2407 https://github.com/STEllAR-GROUP/hpx/pull/2930
2408 https://github.com/STEllAR-GROUP/hpx/pull/2929

Chapter 2. What’s so special about HPX?
- PR #2928 - Fixing execution parameters
- Issue #2927 - CMake warning: ... cycle in constraint graph
- PR #2926 - Default pool rename
- Issue #2925 - Default pool cannot be renamed
- Issue #2924 - hpx:attach-debugger=startup does not work any more
- PR #2923 - Alloc membind
- PR #2922 - This fixes CircleCI errors when running with --hpx:bind=none
- PR #2921 - Custom pool executor was missing priority and stacksize options
- PR #2920 - Adding test to trigger problem reported in #2916
- PR #2919 - Make sure the resource_partitioner is properly destructed on hpx::finalize
- Issue #2918 - hpx::init calls wrong (first) callback when called multiple times
- PR #2917 - Adding util::checkpoint
- Issue #2916 - Weird runtime failures when using a channel and chained continuations
- PR #2915 - Introduce executor parameters customization points
- Issue #2914 - Task assignment to current Pool has unintended consequences
- PR #2913 - Fix rp hang
- PR #2912 - Update contributors
- PR #2911 - Fixing CUDA problems
- PR #2910 - Improve error reporting for process component on POSIX systems
- PR #2909 - Fix typo in include path
- PR #2908 - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2907 - Optionally force-delete remaining channel items on close
- PR #2906 - Making sure generated performance counter names are correct
- Issue #2911 - Improving error reporting for process component on POSIX systems
- PR #2905 - Fix typo in include path
- PR #2904 - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2903 - Optionally force-delete remaining channel items on close
- PR #2902 - Making sure generated performance counter names are correct

---

2.10. Releases

https://github.com/STEllAR-GROUP/hpx/pull/2928
https://github.com/STEllAR-GROUP/hpx/issues/2927
https://github.com/STEllAR-GROUP/hpx/pull/2926
https://github.com/STEllAR-GROUP/hpx/issues/2925
https://github.com/STEllAR-GROUP/hpx/pull/2924
https://github.com/STEllAR-GROUP/hpx/pull/2923
https://github.com/STEllAR-GROUP/hpx/pull/2922
https://github.com/STEllAR-GROUP/hpx/pull/2921
https://github.com/STEllAR-GROUP/hpx/pull/2920
https://github.com/STEllAR-GROUP/hpx/pull/2919
https://github.com/STEllAR-GROUP/hpx/pull/2918
https://github.com/STEllAR-GROUP/hpx/pull/2917
https://github.com/STEllAR-GROUP/hpx/pull/2916
https://github.com/STEllAR-GROUP/hpx/pull/2915
https://github.com/STEllAR-GROUP/hpx/issues/2914
https://github.com/STEllAR-GROUP/hpx/pull/2913
https://github.com/STEllAR-GROUP/hpx/pull/2912
https://github.com/STEllAR-GROUP/hpx/pull/2911
https://github.com/STEllAR-GROUP/hpx/pull/2910
https://github.com/STEllAR-GROUP/hpx/pull/2909
https://github.com/STEllAR-GROUP/hpx/pull/2908
https://github.com/STEllAR-GROUP/hpx/pull/2907
https://github.com/STEllAR-GROUP/hpx/pull/2906
https://github.com/STEllAR-GROUP/hpx/issues/2906
• Issue #2905 - collecting idle-rate performance counters on multiple localities produces an error
• Issue #2904 - build broken for Intel 17 compilers
• PR #2903 - Documentation Updates– Adding New People
• PR #2902 - Fixing service_executor
• PR #2901 - Fixing partitioned_vector creation
• PR #2900 - Add numa-balanced mode to hpx::bind, spread cores over numa domains
• Issue #2899 - hpx::bind does not have a mode that balances cores over numa domains
• PR #2898 - Adding missing #include and missing guard for optional code section
• PR #2897 - Removing dependency on Boost.ICL
• Issue #2896 - Debug build fails without -fpermissive with GCC 7.1 and Boost 1.65
• PR #2895 - Fixing SLURM environment parsing
• PR #2894 - Fix incorrect handling of compile definition with value 0
• Issue #2893 - Disabling schedulers causes build errors
• PR #2892 - added list serializer
• PR #2891 - Resource Partitioner Fixes
• Issue #2890 - Destroying a non-empty channel causes an assertion failure
• PR #2889 - Add check for libatomic
• PR #2888 - Fix compilation problems if HPX_WITH_ITT_NOTIFY=ON
• PR #2887 - Adapt broadcast() to non-unwrapping async<Action>
• PR #2886 - Replace Boost.Random with C++11 <random>
• Issue #2885 - regression in broadcast?
• Issue #2884 - linking -latomic is not portable
• PR #2883 - Explicitly set -pthread flag if available

https://github.com/STEllAR-GROUP/hpx/issues/2905
https://github.com/STEllAR-GROUP/hpx/issues/2904
https://github.com/STEllAR-GROUP/hpx/pull/2903
https://github.com/STEllAR-GROUP/hpx/pull/2902
https://github.com/STEllAR-GROUP/hpx/pull/2901
https://github.com/STEllAR-GROUP/hpx/pull/2900
https://github.com/STEllAR-GROUP/hpx/pull/2899
https://github.com/STEllAR-GROUP/hpx/pull/2898
https://github.com/STEllAR-GROUP/hpx/pull/2897
https://github.com/STEllAR-GROUP/hpx/pull/2896
https://github.com/STEllAR-GROUP/hpx/pull/2895
https://github.com/STEllAR-GROUP/hpx/pull/2894
https://github.com/STEllAR-GROUP/hpx/pull/2893
https://github.com/STEllAR-GROUP/hpx/pull/2892
https://github.com/STEllAR-GROUP/hpx/pull/2891
https://github.com/STEllAR-GROUP/hpx/pull/2890
https://github.com/STEllAR-GROUP/hpx/pull/2889
https://github.com/STEllAR-GROUP/hpx/pull/2888
https://github.com/STEllAR-GROUP/hpx/pull/2887
https://github.com/STEllAR-GROUP/hpx/pull/2886
https://github.com/STEllAR-GROUP/hpx/pull/2885
https://github.com/STEllAR-GROUP/hpx/pull/2884
https://github.com/STEllAR-GROUP/hpx/pull/2883
- PR #2882 - Wrap boost::format uses
- Issue #2881 - hpx not compiling with HPX_WITH_ITTNOTIFY=On
- Issue #2880 - hpx::bind scatter/balanced give wrong pu masks
- PR #2878 - Fix incorrect pool usage masks setup in RP/thread manager
- PR #2877 - Require std::array by default
- PR #2875 - Deprecate use of BOOST_ASSERT
- PR #2874 - Changed serialization of boost.variant to use variadic templates
- Issue #2873 - building with parcelport_mpi fails on cori
- PR #2871 - Adding missing support for throttling scheduler
- PR #2870 - Disambiguate use of base_lco_with_value macros with channel
- Issue #2869 - Difficulty compiling HPX_REGISTER_CHANNEL_DECLARATION(double)
- PR #2868 - Removing unneeded assert
- PR #2867 - Implement parallel::unique
- Issue #2866 - The chunk_size_iterate violates multipass guarantee
- PR #2865 - Only use sched_getcpu on linux machines
- PR #2864 - Create redistribution archive for successful builds
- PR #2863 - Replace casts/assignments with hard-coded memcpy operations
- Issue #2862 - sched_getcpu not available on MacOS
- PR #2861 - Fixing unmatched header defines and recursive inclusion of threadmanager
- Issue #2860 - Master program fails with assertion ‘type == data_type_address’ failed: HPX(assertion_failure)
- Issue #2852 - Support for ARM64
- PR #2858 - Fix misplaced #if #endif’s that cause build failure without THREAD_CUMULATIVE_COUNTS

https://github.com/STEllAR-GROUP/hpx/pull/2882
https://github.com/STEllAR-GROUP/hpx/issues/2881
https://github.com/STEllAR-GROUP/hpx/issues/2880
https://github.com/STEllAR-GROUP/hpx/pull/2878
https://github.com/STEllAR-GROUP/hpx/pull/2877
https://github.com/STEllAR-GROUP/hpx/pull/2875
https://github.com/STEllAR-GROUP/hpx/pull/2874
https://github.com/STEllAR-GROUP/hpx/issues/2873
https://github.com/STEllAR-GROUP/hpx/pull/2871
https://github.com/STEllAR-GROUP/hpx/pull/2870
https://github.com/STEllAR-GROUP/hpx/pull/2869
https://github.com/STEllAR-GROUP/hpx/pull/2868
https://github.com/STEllAR-GROUP/hpx/pull/2867
https://github.com/STEllAR-GROUP/hpx/issues/2866
https://github.com/STEllAR-GROUP/hpx/pull/2865
https://github.com/STEllAR-GROUP/hpx/pull/2864
https://github.com/STEllAR-GROUP/hpx/pull/2863
https://github.com/STEllAR-GROUP/hpx/issues/2862
https://github.com/STEllAR-GROUP/hpx/pull/2861
https://github.com/STEllAR-GROUP/hpx/issues/2860
https://github.com/STEllAR-GROUP/hpx/issues/2852
https://github.com/STEllAR-GROUP/hpx/pull/2858
• PR #2857 - Fix some listing in documentation
• PR #2856 - Fixing component handling for lcos
• PR #2855 - Add documentation for coarrays
• PR #2854 - Support ARM64 in timestamps
• PR #2853 - Update Table 17. Non-modifying Parallel Algorithms in Documentation
• PR #2851 - Allowing for non-default-constructible component types
• PR #2850 - Enable returning future<R> from actions where R is not default-constructible
• PR #2849 - Unify serialization of non-default-constructible types
• Issue #2848 - Components have to be default constructible
• Issue #2847 - Returning a future<R> where R is not default-constructible broken
• Issue #2846 - Unify serialization of non-default-constructible types
• PR #2845 - Add Visual Studio 2015 to the tested toolchains in Appveyor
• Issue #2844 - Change the appveyor build to use the minimal required MSVC version
• Issue #2843 - multi node hello_world hangs
• PR #2842 - Correcting Spelling mistake in docs
• PR #2841 - Fix usage of std::aligned_storage
• PR #2840 - Remove constexpr from a void function
• Issue #2839 - memcpy buffer overflow: load_construct_data() and std::complex members
• Issue #2835 - constexpr functions with void return type break compilation with CUDA 8.0
• Issue #2834 - One suspicion in parallel::detail::handle_exception
• PR #2833 - Implement parallel::merge
• PR #2832 - Fix a strange thing in parallel::util::detail::handle_local_exceptions. (Fix #2818)
• PR #2830 - Break the debugger when a test failed

https://github.com/STEllAR-GROUP/hpx/pull/2857
https://github.com/STEllAR-GROUP/hpx/pull/2856
https://github.com/STEllAR-GROUP/hpx/pull/2855
https://github.com/STEllAR-GROUP/hpx/pull/2854
https://github.com/STEllAR-GROUP/hpx/pull/2853
https://github.com/STEllAR-GROUP/hpx/pull/2851
https://github.com/STEllAR-GROUP/hpx/pull/2850
https://github.com/STEllAR-GROUP/hpx/pull/2849
https://github.com/STEllAR-GROUP/hpx/pull/2845
https://github.com/STEllAR-GROUP/hpx/issues/2848
https://github.com/STEllAR-GROUP/hpx/issues/2847
https://github.com/STEllAR-GROUP/hpx/issues/2846
https://github.com/STEllAR-GROUP/hpx/pull/2845
https://github.com/STEllAR-GROUP/hpx/issues/2844
https://github.com/STEllAR-GROUP/hpx/issues/2843
https://github.com/STEllAR-GROUP/hpx/pull/2842
https://github.com/STEllAR-GROUP/hpx/pull/2841
https://github.com/STEllAR-GROUP/hpx/pull/2840
https://github.com/STEllAR-GROUP/hpx/issues/2839
https://github.com/STEllAR-GROUP/hpx/issues/2835
https://github.com/STEllAR-GROUP/hpx/issues/2834
https://github.com/STEllAR-GROUP/hpx/pull/2833
https://github.com/STEllAR-GROUP/hpx/pull/2832
https://github.com/STEllAR-GROUP/hpx/pull/2830
• Issue #2831\(^\text{2500}\) - parallel/executors/execution_fwd.hpp causes compilation failure in C++11 mode.

• PR #2829\(^\text{2501}\) - Implement an API for asynchronous pack traversal

• PR #2828\(^\text{2502}\) - Split unit test builds on CircleCI to avoid timeouts

• Issue #2827\(^\text{2503}\) - failure to compile hello_world example with -Werror

• PR #2824\(^\text{2504}\) - Making sure promises are marked as started when used as continuations

• PR #2823\(^\text{2505}\) - Add documentation for partitioned_vector_view

• Issue #2822\(^\text{2506}\) - Yet another issue with wait_for similar to #2796

• PR #2821\(^\text{2507}\) - Fix bugs and improve that about HPX_HAVE_CXX11_AUTO_RETURN_VALUE of CMake

• PR #2820\(^\text{2508}\) - Support C++11 in benchmark codes of parallel::partition and parallel::partition_copy

• PR #2819\(^\text{2509}\) - Fix compile errors in unit test of container version of parallel::partition

• Issue #2818\(^\text{2510}\) - A strange thing in parallel::util::detail::handle_local_exceptions

• Issue #2815\(^\text{2511}\) - HPX fails to compile with HPX_WITH_CUDA=ON and the new CUDA 9.0 RC

• Issue #2814\(^\text{2512}\) - Using ‘gmakeN’ after ‘cmake’ produces error in src/CMakeFiles/hpx.dir/runtime/agas/addressing_service.cpp.o

• PR #2813\(^\text{2513}\) - Properly support [[noreturn]] attribute if available

• Issue #2812\(^\text{2514}\) - Compilation fails with gcc 7.1.1

• PR #2811\(^\text{2515}\) - Adding hpx::launch::lazy and support for async, dataflow, and future::then

• PR #2810\(^\text{2516}\) - Add option allowing to disable deprecation warning

• PR #2809\(^\text{2517}\) - Disable throttling scheduler if HWLOC is not found/used

• PR #2808\(^\text{2518}\) - Fix compile errors on some environments of parallel::partition

• Issue #2807\(^\text{2519}\) - Difficulty building with HPX_WITH_HWLOC=Off

• PR #2806\(^\text{2520}\) - Partitioned vector

• PR #2805\(^\text{2521}\) - Serializing collections with non-default constructible data

---

\(^{2500}\) https://github.com/STEllAR-GROUP/hpx/issues/2831

\(^{2501}\) https://github.com/STEllAR-GROUP/hpx/pull/2829

\(^{2502}\) https://github.com/STEllAR-GROUP/hpx/pull/2828

\(^{2503}\) https://github.com/STEllAR-GROUP/hpx/issues/2827

\(^{2504}\) https://github.com/STEllAR-GROUP/hpx/pull/2824

\(^{2505}\) https://github.com/STEllAR-GROUP/hpx/pull/2823

\(^{2506}\) https://github.com/STEllAR-GROUP/hpx/issues/2822

\(^{2507}\) https://github.com/STEllAR-GROUP/hpx/pull/2821

\(^{2508}\) https://github.com/STEllAR-GROUP/hpx/pull/2820

\(^{2509}\) https://github.com/STEllAR-GROUP/hpx/pull/2819

\(^{2510}\) https://github.com/STEllAR-GROUP/hpx/issues/2818

\(^{2511}\) https://github.com/STEllAR-GROUP/hpx/issues/2815

\(^{2512}\) https://github.com/STEllAR-GROUP/hpx/issues/2814

\(^{2513}\) https://github.com/STEllAR-GROUP/hpx/pull/2813

\(^{2514}\) https://github.com/STEllAR-GROUP/hpx/issues/2812

\(^{2515}\) https://github.com/STEllAR-GROUP/hpx/pull/2811

\(^{2516}\) https://github.com/STEllAR-GROUP/hpx/pull/2810

\(^{2517}\) https://github.com/STEllAR-GROUP/hpx/pull/2809

\(^{2518}\) https://github.com/STEllAR-GROUP/hpx/pull/2808

\(^{2519}\) https://github.com/STEllAR-GROUP/hpx/issues/2807

\(^{2520}\) https://github.com/STEllAR-GROUP/hpx/pull/2806

\(^{2521}\) https://github.com/STEllAR-GROUP/hpx/pull/2805
• PR #2802 - Fix FreeBSD 11
  • Issue #2801 - Rate limiting techniques in io_service
  • Issue #2800 - New Launch Policy: async_if
• PR #2799 - Fix a unit test failure on GCC in tuple_cat
• PR #2798 - bump minimum required cmake to 3.0 in test
• PR #2797 - Making sure future::wait_for et.al. work properly for action results
• Issue #2796 - wait_for does always in “deferred” state for calls on remote localities
• Issue #2795 - Serialization of types without default constructor
• PR #2794 - Fixing test for partitioned_vector iteration
• PR #2792 - Implemented segmented find and its variations for partitioned vector
• PR #2791 - Circumvent scary warning about placement new
• PR #2790 - Fix OSX build
• PR #2789 - Resource partitioner
• PR #2788 - Adapt parallel::is_heap and parallel::is_heap_until to Ranges TS
• PR #2787 - Unwrap hotfixes
• PR #2786 - Update CMake Minimum Version to 3.3.2 (refs #2565)
• Issue #2785 - Issues with masks and cpuset
• PR #2784 - Error with reduce and transform reduce fixed
• PR #2783 - StackOverflow integration with libsigsegv
• PR #2782 - Replace boost::atomic with std::atomic (where possible)
• PR #2781 - Check for and optionally use [[deprecated]] attribute
• PR #2780 - Adding empty (but non-trivial) destructor to circumvent warnings
• PR #2779 - Exception info tweaks

2522 https://github.com/STEllAR-GROUP/hpx/pull/2802
2523 https://github.com/STEllAR-GROUP/hpx/issues/2801
2524 https://github.com/STEllAR-GROUP/hpx/issues/2800
2525 https://github.com/STEllAR-GROUP/hpx/pull/2799
2526 https://github.com/STEllAR-GROUP/hpx/pull/2798
2527 https://github.com/STEllAR-GROUP/hpx/pull/2797
2528 https://github.com/STEllAR-GROUP/hpx/issues/2796
2529 https://github.com/STEllAR-GROUP/hpx/issues/2795
2530 https://github.com/STEllAR-GROUP/hpx/pull/2794
2531 https://github.com/STEllAR-GROUP/hpx/pull/2792
2532 https://github.com/STEllAR-GROUP/hpx/pull/2791
2533 https://github.com/STEllAR-GROUP/hpx/pull/2790
2534 https://github.com/STEllAR-GROUP/hpx/pull/2789
2535 https://github.com/STEllAR-GROUP/hpx/pull/2788
2536 https://github.com/STEllAR-GROUP/hpx/pull/2787
2537 https://github.com/STEllAR-GROUP/hpx/pull/2786
2538 https://github.com/STEllAR-GROUP/hpx/issues/2785
2539 https://github.com/STEllAR-GROUP/hpx/pull/2794
2540 https://github.com/STEllAR-GROUP/hpx/pull/2792
2541 https://github.com/STEllAR-GROUP/hpx/pull/2791
2542 https://github.com/STEllAR-GROUP/hpx/pull/2790
2543 https://github.com/STEllAR-GROUP/hpx/pull/2789
• PR #2778 - Implement parallel::partition
• PR #2777 - Improve error handling in gather_here/gather_there
• PR #2776 - Fix a bug in compiler version check
• PR #2775 - Fix compilation when HPX_WITH_LOGGING is OFF
• PR #2774 - Removing dependency on Boost.Date_Time
• PR #2773 - Add sync_images() method to spmd_block class
• PR #2772 - Adding documentation for PAPI counters
• PR #2771 - Removing boost preprocessor dependency
• PR #2770 - Adding test, fixing deadlock in config registry
• PR #2769 - Remove some other warnings and errors detected by clang 5.0
• Issue #2768 - Is there iterator tag for HPX?
• PR #2767 - Improvements to continuation annotation
• PR #2765 - gcc split stack support for HPX threads #620
• PR #2764 - Fix some uses of begin/end, remove unnecessary includes
• PR #2763 - Bump minimal Boost version to 1.55.0
• PR #2762 - hpx::partitioned_vector serializer
• PR #2761 - Adding configuration summary to cmake output and –hpx:info
• PR #2760 - Removing 1d_hydro example as it is broken
• PR #2758 - Remove various warnings detected by clang 5.0
• Issue #2757 - In case of a “raw thread” is needed per core for implementing parallel algorithm, what is good practice in HPX?
• PR #2756 - Allowing for LCOs to be simple components
• PR #2755 - Removing make_index_pack_unrolled
• PR #2754 - Implement parallel::unique_copy

2.10. Releases

https://github.com/STEllAR-GROUP/hpx/pull/2778
https://github.com/STEllAR-GROUP/hpx/pull/2777
https://github.com/STEllAR-GROUP/hpx/pull/2776
https://github.com/STEllAR-GROUP/hpx/pull/2775
https://github.com/STEllAR-GROUP/hpx/pull/2774
https://github.com/STEllAR-GROUP/hpx/pull/2773
https://github.com/STEllAR-GROUP/hpx/pull/2772
https://github.com/STEllAR-GROUP/hpx/pull/2771
https://github.com/STEllAR-GROUP/hpx/pull/2770
https://github.com/STEllAR-GROUP/hpx/pull/2769
https://github.com/STEllAR-GROUP/hpx/pull/2768
https://github.com/STEllAR-GROUP/hpx/pull/2767
https://github.com/STEllAR-GROUP/hpx/pull/2765
https://github.com/STEllAR-GROUP/hpx/pull/2764
https://github.com/STEllAR-GROUP/hpx/pull/2763
https://github.com/STEllAR-GROUP/hpx/pull/2762
https://github.com/STEllAR-GROUP/hpx/pull/2761
https://github.com/STEllAR-GROUP/hpx/pull/2760
https://github.com/STEllAR-GROUP/hpx/pull/2758
https://github.com/STEllAR-GROUP/hpx/pull/2757
https://github.com/STEllAR-GROUP/hpx/pull/2756
https://github.com/STEllAR-GROUP/hpx/pull/2755
https://github.com/STEllAR-GROUP/hpx/pull/2754
• PR #2753 - Fixing detection of [[fallthrough]] attribute
• PR #2752 - New thread priority names
• PR #2751 - Replace boost::exception with proposed exception_info
• PR #2750 - Replace boost::iterator_range
• PR #2749 - Fixing hdf5 examples
• Issue #2748 - HPX fails to build with enabled hdf5 examples
• Issue #2747 - Inherited task priorities break certain DAG optimizations
• Issue #2746 - HPX segfaulting with valgrind
• PR #2745 - Adding extended arithmetic performance counters
• PR #2744 - Adding ability to statistics counters to reset base counter
• Issue #2743 - Statistics counter does not support resetting
• PR #2742 - Making sure Vc V2 builds without additional HPX configuration flags
• PR #2741 - Deprecate unwrapped and implement unwrap and unwrapping
• PR #2740 - Coroutine stackoverflow detection for linux/posix; Issue #2408
• PR #2739 - Add files via upload
• PR #2738 - Appveyor support
• PR #2737 - Fixing 2735
• Issue #2736 - 1d_hydro example doesn’t work
• Issue #2735 - partitioned_vectorSubview test failing
• PR #2734 - Add C++11 range utilities
• PR #2733 - Adapting iterator requirements for parallel algorithms
• PR #2732 - Integrate C++ Co-arrays
• PR #2731 - Adding on_migrated event handler to migratable component instances

https://github.com/STEllAR-GROUP/hpx/pull/2753
https://github.com/STEllAR-GROUP/hpx/pull/2752
https://github.com/STEllAR-GROUP/hpx/pull/2751
https://github.com/STEllAR-GROUP/hpx/pull/2750
https://github.com/STEllAR-GROUP/hpx/pull/2749
https://github.com/STEllAR-GROUP/hpx/issues/2748
https://github.com/STEllAR-GROUP/hpx/issues/2747
https://github.com/STEllAR-GROUP/hpx/issues/2746
https://github.com/STEllAR-GROUP/hpx/pull/2745
https://github.com/STEllAR-GROUP/hpx/pull/2744
https://github.com/STEllAR-GROUP/hpx/issues/2743
https://github.com/STEllAR-GROUP/hpx/pull/2740
https://github.com/STEllAR-GROUP/hpx/pull/2739
https://github.com/STEllAR-GROUP/hpx/pull/2738
https://github.com/STEllAR-GROUP/hpx/pull/2737
https://github.com/STEllAR-GROUP/hpx/pull/2736
https://github.com/STEllAR-GROUP/hpx/issues/2735
https://github.com/STEllAR-GROUP/hpx/pull/2734
https://github.com/STEllAR-GROUP/hpx/pull/2733
https://github.com/STEllAR-GROUP/hpx/pull/2732
https://github.com/STEllAR-GROUP/hpx/pull/2731
- Issue #2729 - Add on_migrated() event handler to migratable components
- Issue #2728 - Why Projection is needed in parallel algorithms?
- PR #2727 - Cmake files for StackOverflow Detection
- PR #2726 - CMake for Stack Overflow Detection
- PR #2725 - Implemented segmented algorithms for partitioned vector
- PR #2724 - Fix examples in Action documentation
- PR #2723 - Enable lcos::channel<T>::register_as
- Issue #2722 - channel register_as() failing on compilation
- PR #2721 - Mind map
- PR #2720 - reorder forward declarations to get rid of C++14-only auto return types
- PR #2719 - Add documentation for partitioned_vector and add features in pack.hpp
- Issue #2718 - Some forward declarations in execution_fwd.hpp aren’t C++11-compatible
- PR #2717 - Config support for fallthrough attribute
- PR #2716 - Implement parallel::partition_copy
- PR #2715 - initial import of icu string serializer
- PR #2714 - initial import of valarray serializer
- PR #2713 - Remove slashes before CMAKE_FILES_DIRECTORY variables
- PR #2712 - Fixing wait for 1751
- PR #2711 - Adjust code for minimal supported GCC having being bumped to 4.9
- PR #2710 - Adding code of conduct
- PR #2709 - Fixing UB in destroy tests
- PR #2708 - Add inline to prevent multiple definition issue
- Issue #2707 - Multiple defined symbols for task_block.hpp in VS2015

https://github.com/STEllAR-GROUP/hpx/issues/2729
https://github.com/STEllAR-GROUP/hpx/issues/2728
https://github.com/STEllAR-GROUP/hpx/pull/2727
https://github.com/STEllAR-GROUP/hpx/pull/2726
https://github.com/STEllAR-GROUP/hpx/pull/2725
https://github.com/STEllAR-GROUP/hpx/pull/2724
https://github.com/STEllAR-GROUP/hpx/pull/2723
https://github.com/STEllAR-GROUP/hpx/pull/2722
https://github.com/STEllAR-GROUP/hpx/pull/2721
https://github.com/STEllAR-GROUP/hpx/pull/2720
https://github.com/STEllAR-GROUP/hpx/pull/2719
https://github.com/STEllAR-GROUP/hpx/pull/2718
https://github.com/STEllAR-GROUP/hpx/pull/2717
https://github.com/STEllAR-GROUP/hpx/pull/2716
https://github.com/STEllAR-GROUP/hpx/pull/2715
https://github.com/STEllAR-GROUP/hpx/pull/2714
https://github.com/STEllAR-GROUP/hpx/pull/2713
https://github.com/STEllAR-GROUP/hpx/pull/2712
https://github.com/STEllAR-GROUP/hpx/pull/2711
https://github.com/STEllAR-GROUP/hpx/pull/2710
https://github.com/STEllAR-GROUP/hpx/pull/2709
https://github.com/STEllAR-GROUP/hpx/pull/2708
https://github.com/STEllAR-GROUP/hpx/pull/2707

2.10. Releases
• PR #2706 - Adding .clang-format file
• PR #2704 - Add a synchronous mapping API
• Issue #2703 - Request: Add the .clang-format file to the repository
• Issue #2702 - STEllAR-GROUP/Vc slower than VCv1 possibly due to wrong instructions generated
• Issue #2701 - Datapar with STEllAR-GROUP/Vc requires obscure flag
• Issue #2700 - Naming inconsistency in parallel algorithms
• Issue #2699 - Iterator requirements are different from standard in parallel copy_if.
• PR #2698 - Properly releasing parcelport write handlers
• Issue #2697 - Compile error in addressing_service.cpp
• Issue #2696 - Building and using HPX statically: undefined references from runtime_support_server.cpp
• Issue #2695 - Executor changes cause compilation failures
• PR #2694 - Refining C++ language mode detection for MSVC
• PR #2693 - P0443 r2
• PR #2692 - Partially reverting changes to parcel_await
• Issue #2689 - HPX build fails when HPX_WITH_CUDA is enabled
• PR #2688 - Make Cuda Clang builds pass
• PR #2687 - Add an is_tuple_like trait for sequenceable type detection
• PR #2686 - Allowing throttling scheduler to be used without idle backoff
• PR #2685 - Add support of std::array to hpx::util::tuple_size and tuple_element
• PR #2684 - Adding new statistics performance counters
• PR #2683 - Replace boost::exception_ptr with std::exception_ptr
• Issue #2682 - HPX does not compile with HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF
• PR #2681 - Attempt to fix problem in managed_component_base

https://github.com/STEllAR-GROUP/hpx/pull/2706
https://github.com/STEllAR-GROUP/hpx/pull/2704
https://github.com/STEllAR-GROUP/hpx/issues/2703
https://github.com/STEllAR-GROUP/hpx/issues/2702
https://github.com/STEllAR-GROUP/hpx/issues/2701
https://github.com/STEllAR-GROUP/hpx/issues/2700
https://github.com/STEllAR-GROUP/hpx/pull/2698
https://github.com/STEllAR-GROUP/hpx/pull/2697
https://github.com/STEllAR-GROUP/hpx/pull/2696
https://github.com/STEllAR-GROUP/hpx/pull/2695
https://github.com/STEllAR-GROUP/hpx/pull/2694
https://github.com/STEllAR-GROUP/hpx/pull/2693
https://github.com/STEllAR-GROUP/hpx/pull/2692
https://github.com/STEllAR-GROUP/hpx/pull/2691
- PR #2680 - Fix bad size during archive creation
- Issue #2679 - Mismatch between size of archive and container
- Issue #2678 - In parallel algorithm, other tasks are executed to the end even if an exception occurs in any task.
- PR #2677 - Adding include check for std::addressof
- PR #2676 - Adding parallel::destroy and destroy_n
- PR #2675 - Making sure statistics counters work as expected
- PR #2674 - Turning assertions into exceptions
- PR #2673 - Inhibit direct conversion from future<future<T>> -> future<void>
- PR #2672 - C++17 invoke forms
- PR #2671 - Adding uninitialized_value_construct and uninitialized_value_construct_n
- PR #2670 - Integrate spmd multidimensional views for partitioned_vectors
- PR #2669 - Adding uninitialized_default_construct and uninitialized_default_construct_n
- PR #2668 - Fixing documentation index
- Issue #2667 - Ambiguity of nested hpx::future<void>'s
- Issue #2666 - Statistics Performance counter is not working
- PR #2664 - Adding uninitialized_move and uninitialized_move_n
- Issue #2663 - Seg fault in managed_component::get_base_gid, possibly cause by util::reinitializable_static
- Issue #2662 - Crash in managed_component::get_base_gid due to problem with util::reinitializable_static
- PR #2665 - Hide the detail namespace in doxygen per default
- PR #2660 - Add documentation to hpx::util::unwrapped and hpx::util::unwrapped2
- PR #2659 - Improve integration with vcpkg
- PR #2658 - Unify access_data trait for use in both, serialization and de-serialization
- PR #2657 - Removing hpx::lcos::queue<T>

https://github.com/STEllAR-GROUP/hpx/pull/2680
https://github.com/STEllAR-GROUP/hpx/issues/2679
https://github.com/STEllAR-GROUP/hpx/issues/2678
https://github.com/STEllAR-GROUP/hpx/pull/2677
https://github.com/STEllAR-GROUP/hpx/pull/2676
https://github.com/STEllAR-GROUP/hpx/pull/2675
https://github.com/STEllAR-GROUP/hpx/pull/2674
https://github.com/STEllAR-GROUP/hpx/pull/2673
https://github.com/STEllAR-GROUP/hpx/pull/2672
https://github.com/STEllAR-GROUP/hpx/pull/2671
https://github.com/STEllAR-GROUP/hpx/pull/2670
https://github.com/STEllAR-GROUP/hpx/pull/2669
https://github.com/STEllAR-GROUP/hpx/pull/2668
https://github.com/STEllAR-GROUP/hpx/issues/2667
https://github.com/STEllAR-GROUP/hpx/pull/2666
https://github.com/STEllAR-GROUP/hpx/pull/2664
https://github.com/STEllAR-GROUP/hpx/issues/2663
https://github.com/STEllAR-GROUP/hpx/issues/2662
https://github.com/STEllAR-GROUP/hpx/pull/2665
https://github.com/STEllAR-GROUP/hpx/pull/2660
https://github.com/STEllAR-GROUP/hpx/pull/2659
https://github.com/STEllAR-GROUP/hpx/pull/2658
https://github.com/STEllAR-GROUP/hpx/pull/2657
• PR #2656 - Reduce MAX_TERMINATED_THREADS default, improve memory use on manycore cpus
• PR #2655 - Maintenance for emulate-deleted macros
• PR #2654 - Implement parallel is_heap and is_heap_until
• PR #2653 - Drop support for VS2013
• PR #2652 - This patch makes sure that all parcels in a batch are properly handled
• PR #2649 - Update docs (Table 18) - move transform to end
• Issue #2647 - hpx::parcelset::detail::parcel_data::has_continuation_ is uninitialized
• Issue #2644 - Some .vcxproj in the HPX.sln fail to build
• Issue #2641 - hpx::lcos::queue should be deprecated
• PR #2640 - A new throttling policy with public APIs to suspend/resume
• PR #2639 - Fix a tiny typo in tutorial.
• Issue #2638 - Invalid return type ‘void’ of constexpr function
• PR #2636 - Add and use HPX_MSVC_WARNING_PRAGMA for #pragma warning
• PR #2633 - Distributed define_spmd_block
• PR #2632 - Making sure container serialization uses size-compatible types
• PR #2631 - Add lcos::local::one_element_channel
• PR #2629 - Move unordered_map out of parcelport into hpx/concurrent
• PR #2628 - Making sure that shutdown does not hang
• PR #2627 - Fix serialization
• PR #2626 - Generate cmake_variables.qbk and cmake_toolchains.qbk outside of the source tree
• PR #2625 - Supporting -std=c++17 flag
• PR #2624 - Fixing a small cmake typo
• PR #2622 - Update CMake minimum required version to 3.0.2 (closes #2621)
• Issue #2621 - Compiling hpx master fails with /usr/bin/ld: final link failed: Bad value
• PR #2620 - Remove warnings due to some captured variables
• PR #2619 - LF multiple parcels
• PR #2618 - Some fixes to libfabric that didn’t get caught before the merge
• PR #2617 - Adding hpx::local_new
• PR #2616 - Documentation: Extract all entities in order to autolink functions correctly
• Issue #2615 - Documentation: Linking functions is broken
• PR #2614 - Adding serialization for std::deque
• PR #2613 - We need to link with boost.thread and boost.chrono if we use boost.context
• PR #2612 - Making sure for_loop_n(par, . . . ) is actually executed in parallel
• PR #2611 - Add documentation to invoke_fused and friends NFC
• PR #2610 - Added reduction templates using an identity value
• PR #2609 - Fixing some unused vars in inspect
• PR #2607 - Fixed build for mingw
• PR #2606 - Supporting generic context for boost >= 1.61
• PR #2605 - Parcelport libfabric3
• PR #2604 - Adding allocator support to promise and friends
• PR #2603 - Barrier hang
• PR #2602 - Changes to scheduler to steal from one high-priority queue
• Issue #2601 - High priority tasks are not executed first
• PR #2600 - Compat fixes
• PR #2599 - Compatibility layer for threading support
• PR #2598 - V1.1

2.10. Releases

2683 https://github.com/STEllAR-GROUP/hpx/issues/2621
2684 https://github.com/STEllAR-GROUP/hpx/pull/2620
2685 https://github.com/STEllAR-GROUP/hpx/pull/2619
2686 https://github.com/STEllAR-GROUP/hpx/pull/2618
2687 https://github.com/STEllAR-GROUP/hpx/pull/2617
2688 https://github.com/STEllAR-GROUP/hpx/pull/2616
2689 https://github.com/STEllAR-GROUP/hpx/issues/2615
2690 https://github.com/STEllAR-GROUP/hpx/pull/2614
2691 https://github.com/STEllAR-GROUP/hpx/pull/2613
2692 https://github.com/STEllAR-GROUP/hpx/pull/2612
2693 https://github.com/STEllAR-GROUP/hpx/pull/2611
2694 https://github.com/STEllAR-GROUP/hpx/pull/2610
2695 https://github.com/STEllAR-GROUP/hpx/pull/2608
2696 https://github.com/STEllAR-GROUP/hpx/pull/2607
2697 https://github.com/STEllAR-GROUP/hpx/pull/2606
2698 https://github.com/STEllAR-GROUP/hpx/pull/2605
2699 https://github.com/STEllAR-GROUP/hpx/pull/2604
2700 https://github.com/STEllAR-GROUP/hpx/pull/2603
2701 https://github.com/STEllAR-GROUP/hpx/pull/2602
2702 https://github.com/STEllAR-GROUP/hpx/issues/2601
2703 https://github.com/STEllAR-GROUP/hpx/pull/2600
2704 https://github.com/STEllAR-GROUP/hpx/pull/2599
2705 https://github.com/STEllAR-GROUP/hpx/pull/2598
• PR #2597 - Release V1.0
• PR #2592 - First attempt to introduce spmd_block in hpx
• PR #2586 - local_segment in segmented_iterator_traits
• Issue #2584 - Add allocator support to promise, packaged_task and friends
• PR #2576 - Add missing dependencies of cuda based tests
• PR #2575 - Remove warnings due to some captured variables
• Issue #2574 - MSVC 2015 Compiler crash when building HPX
• Issue #2568 - Remove throttle_scheduler as it has been abandoned
• Issue #2566 - Add an inline versioning namespace before 1.0 release
• Issue #2565 - Raise minimal cmake version requirement
• PR #2556 - Fixing scan partitioner
• PR #2546 - Broadcast async
• Issue #2543 - make install fails due to a non-existing .so file
• PR #2495 - wait_or_add_new returning thread_id_type
• Issue #2480 - Unable to register new performance counter
• Issue #2471 - no type named 'fcontext_t' in namespace
• Issue #2456 - Re-implement hpx::util::unwrapped
• Issue #2455 - Add more arithmetic performance counters
• PR #2454 - Fix a couple of warnings and compiler errors
• PR #2453 - Timed executor support
• PR #2447 - Implementing new executor API (P0443)
• Issue #2439 - Implement executor proposal
• Issue #2408 - Stackoverflow detection for linux, e.g. based on libsigsegv

https://github.com/STEllAR-GROUP/hpx/pull/2597
https://github.com/STEllAR-GROUP/hpx/pull/2592
https://github.com/STEllAR-GROUP/hpx/pull/2586
https://github.com/STEllAR-GROUP/hpx/issues/2584
https://github.com/STEllAR-GROUP/hpx/pull/2576
https://github.com/STEllAR-GROUP/hpx/pull/2575
https://github.com/STEllAR-GROUP/hpx/issues/2574
https://github.com/STEllAR-GROUP/hpx/issues/2568
https://github.com/STEllAR-GROUP/hpx/issues/2566
https://github.com/STEllAR-GROUP/hpx/issues/2565
https://github.com/STEllAR-GROUP/hpx/pull/2556
https://github.com/STEllAR-GROUP/hpx/pull/2546
https://github.com/STEllAR-GROUP/hpx/issues/2543
https://github.com/STEllAR-GROUP/hpx/pull/2495
https://github.com/STEllAR-GROUP/hpx/issues/2480
https://github.com/STEllAR-GROUP/hpx/issues/2471
https://github.com/STEllAR-GROUP/hpx/issues/2456
https://github.com/STEllAR-GROUP/hpx/issues/2455
https://github.com/STEllAR-GROUP/hpx/pull/2454
https://github.com/STEllAR-GROUP/hpx/pull/2453
https://github.com/STEllAR-GROUP/hpx/pull/2447
https://github.com/STEllAR-GROUP/hpx/issues/2439
https://github.com/STEllAR-GROUP/hpx/issues/2408
• PR #2377\textsuperscript{2729} - Add a customization point for put_parcel so we can override actions
• Issue #2368\textsuperscript{2730} - HPX_ASSERT problem
• Issue #2324\textsuperscript{2731} - Change default number of threads used to the maximum of the system
• Issue #2266\textsuperscript{2732} - hpx_0.9.99 make tests fail
• PR #2195\textsuperscript{2733} - Support for code completion in VIM
• Issue #2137\textsuperscript{2734} - Hpx does not compile over osx
• Issue #2092\textsuperscript{2735} - make tests should just build the tests
• Issue #2026\textsuperscript{2736} - Build HPX with Apple’s clang
• Issue #1932\textsuperscript{2737} - hpx with PBS fails on multiple localities
• PR #1914\textsuperscript{2738} - Parallel heap algorithm implementations WIP
• Issue #1598\textsuperscript{2739} - Disconnecting a locality results in segfault using heartbeat example
• Issue #1404\textsuperscript{2740} - unwrapped doesn’t work with movable only types
• Issue #1400\textsuperscript{2741} - hpx::util::unwrapped doesn’t work with non-future types
• Issue #1205\textsuperscript{2742} - TSS is broken
• Issue #1126\textsuperscript{2743} - vector&lt;future&lt;T&gt; &gt; does not work gracefully with dataflow, when_all and unwrapped
• Issue #1056\textsuperscript{2744} - Thread manager cleanup
• Issue #863\textsuperscript{2745} - Futures should not require a default constructor
• Issue #856\textsuperscript{2746} - Allow runtimemode_connect to be used with security enabled
• Issue #726\textsuperscript{2747} - Valgrind
• Issue #701\textsuperscript{2748} - Add RCR performance counter component
• Issue #528\textsuperscript{2749} - Add support for known failures and warning count/comparisons to hpx_run_tests.py

\textsuperscript{2729} https://github.com/STElAR-GROUP/hpx/pull/2377
\textsuperscript{2730} https://github.com/STElAR-GROUP/hpx/issues/2368
\textsuperscript{2731} https://github.com/STElAR-GROUP/hpx/issues/2324
\textsuperscript{2732} https://github.com/STElAR-GROUP/hpx/issues/2266
\textsuperscript{2733} https://github.com/STElAR-GROUP/hpx/pull/2195
\textsuperscript{2734} https://github.com/STElAR-GROUP/hpx/issues/2137
\textsuperscript{2735} https://github.com/STElAR-GROUP/hpx/issues/2092
\textsuperscript{2736} https://github.com/STElAR-GROUP/hpx/issues/2026
\textsuperscript{2737} https://github.com/STElAR-GROUP/hpx/issues/1932
\textsuperscript{2738} https://github.com/STElAR-GROUP/hpx/pull/1914
\textsuperscript{2739} https://github.com/STElAR-GROUP/hpx/issues/1598
\textsuperscript{2740} https://github.com/STElAR-GROUP/hpx/issues/1404
\textsuperscript{2741} https://github.com/STElAR-GROUP/hpx/issues/1400
\textsuperscript{2742} https://github.com/STElAR-GROUP/hpx/issues/1205
\textsuperscript{2743} https://github.com/STElAR-GROUP/hpx/issues/1126
\textsuperscript{2744} https://github.com/STElAR-GROUP/hpx/issues/1056
\textsuperscript{2745} https://github.com/STElAR-GROUP/hpx/issues/863
\textsuperscript{2746} https://github.com/STElAR-GROUP/hpx/issues/856
\textsuperscript{2747} https://github.com/STElAR-GROUP/hpx/issues/726
\textsuperscript{2748} https://github.com/STElAR-GROUP/hpx/issues/701
\textsuperscript{2749} https://github.com/STElAR-GROUP/hpx/issues/528
2.10.12 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 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.
- 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 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 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 old 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 improved 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.

2750 https://github.com/STEllAR-GROUP/hpx/pull/2289
2751 https://github.com/STEllAR-GROUP/hpx/issues/2333
• 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.

• 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 #25962753 - Adding apex data
- PR #25952754 - Remove obsolete file
- Issue #25942755 - FindOpenCL.cmake mismatch with the official cmake module
- PR #25922756 - First attempt to introduce spmd_block in hpx
- Issue #25912757 - Feature request: continuation (then) which does not require the callable object to take a future<R> as parameter
- PR #25882758 - Daint fixes
- PR #25872759 - Fixing transfer_(continuation)_action::schedule
- PR #25852760 - Work around MSVC having an ICE when compiling with -Ob2
- PR #25832761 - changing 7zip command to 7za in roll_release.sh
- PR #25822762 - First attempt to introduce spmd_block in hpx
- PR #25812763 - Enable annotated function for parallel algorithms

2753 https://github.com/STEllAR-GROUP/hpx/pull/2596
2754 https://github.com/STEllAR-GROUP/hpx/pull/2595
2755 https://github.com/STEllAR-GROUP/hpx/issues/2594
2756 https://github.com/STEllAR-GROUP/hpx/pull/2592
2757 https://github.com/STEllAR-GROUP/hpx/issues/2591
2758 https://github.com/STEllAR-GROUP/hpx/pull/2588
2759 https://github.com/STEllAR-GROUP/hpx/pull/2587
2760 https://github.com/STEllAR-GROUP/hpx/pull/2585
2761 https://github.com/STEllAR-GROUP/hpx/pull/2583
2762 https://github.com/STEllAR-GROUP/hpx/pull/2582
2763 https://github.com/STEllAR-GROUP/hpx/pull/2581
• PR #2580\textsuperscript{2764} - First attempt to introduce spmd\_block in hpx
• PR #2579\textsuperscript{2765} - Make thread NICE level setting an option
• PR #2578\textsuperscript{2766} - Implementing enqueue instead of busy wait when no sender is available
• PR #2577\textsuperscript{2767} - Retrieve -std=c++11 consistent nvcc flag
• PR #2576\textsuperscript{2768} - Add missing dependencies of cuda based tests
• PR #2575\textsuperscript{2769} - Remove warnings due to some captured variables
• PR #2573\textsuperscript{2770} - Attempt to resolve resolve\_locality
• PR #2572\textsuperscript{2771} - Adding APEX hooks to background thread
• PR #2571\textsuperscript{2772} - Pick up hpx\_ignore\_batch\_env from config map
• PR #2570\textsuperscript{2773} - Add commandline options –hpx:print-counters-locally
• PR #2569\textsuperscript{2774} - Fix computeapi unit tests
• PR #2567\textsuperscript{2775} - This adds another barrier:|synchronize before registering performance counters
• PR #2564\textsuperscript{2776} - Cray static toolchain support
• PR #2563\textsuperscript{2777} - Fixed unhandled exception during startup
• PR #2562\textsuperscript{2778} - Remove partitioned\_vector\_cu from build tree when nvcc is used
• Issue #2561\textsuperscript{2779} - octo-tiger crash with commit 6e921495ff6c26f125d62629cbaad0525f14f7ab
• PR #2560\textsuperscript{2780} - Prevent -Wundef warnings on Vc version checks
• PR #2559\textsuperscript{2781} - Allowing CUDA callback to set the future directly from an OS thread
• PR #2558\textsuperscript{2782} - Remove warnings due to float precisions
• PR #2557\textsuperscript{2783} - Removing bogus handling of compile flags for CUDA
• PR #2556\textsuperscript{2784} - Fixing scan partitioner
• PR #2554\textsuperscript{2785} - Add more diagnostics to error thrown from find\_appropriate\_destination
• Issue #2555\textsuperscript{2786} - No valid parcelport configured
• PR #2553 - Add cmake cuda_arch option
• PR #2552 - Remove incomplete datapar bindings to libflatarray
• PR #2551 - Rename hwloc_topology to hwloc_topology_info
• PR #2550 - Apex api updates
• PR #2549 - Pre-include defines.hpp to get the macro HPX_HAVE_CUDA value
• PR #2548 - Fixing issue with disconnect
• PR #2546 - Some fixes around cuda clang partitioned_vector example
• PR #2545 - Fix uses of the Vc2 datapar flags; the value, not the type, should be passed to functions
• PR #2542 - Make HPX_WITH_MALLOC easier to use
• PR #2541 - avoid recompiles when enabling/disabling examples
• PR #2540 - Fixing usage of target_link_libraries()
• PR #2539 - fix RPATH behaviour
• Issue #2538 - HPX_WITH_CUDA corrupts compilation flags
• PR #2537 - Add output of a Bazel Skylark extension for paths and compile options
• PR #2536 - Add counter exposing total available memory to Windows as well
• PR #2535 - Remove obsolete support for security
• Issue #2534 - Remove command line option --hpx:run-agas-server
• PR #2533 - Pre-cache locality endpoints during bootstrap
• PR #2532 - Fixing handling of GIDs during serialization preprocessing
• PR #2531 - Amend uses of the term “functor”
• PR #2530 - added counter for reading available memory
• PR #2529 - Facilities to create actions from lambdas
• PR #2528 - Updated docs: HPX_WITH_EXAMPLES

https://github.com/STEllAR-GROUP/hpx/pull/2553
https://github.com/STEllAR-GROUP/hpx/pull/2552
https://github.com/STEllAR-GROUP/hpx/pull/2551
https://github.com/STEllAR-GROUP/hpx/pull/2550
https://github.com/STEllAR-GROUP/hpx/pull/2549
https://github.com/STEllAR-GROUP/hpx/pull/2548
https://github.com/STEllAR-GROUP/hpx/pull/2547
https://github.com/STEllAR-GROUP/hpx/pull/2546
https://github.com/STEllAR-GROUP/hpx/pull/2545
https://github.com/STEllAR-GROUP/hpx/pull/2542
https://github.com/STEllAR-GROUP/hpx/pull/2541
https://github.com/STEllAR-GROUP/hpx/pull/2540
https://github.com/STEllAR-GROUP/hpx/pull/2539
https://github.com/STEllAR-GROUP/hpx/issues/2538
https://github.com/STEllAR-GROUP/hpx/pull/2537
https://github.com/STEllAR-GROUP/hpx/pull/2536
https://github.com/STEllAR-GROUP/hpx/pull/2535
https://github.com/STEllAR-GROUP/hpx/issues/2534
https://github.com/STEllAR-GROUP/hpx/pull/2533
https://github.com/STEllAR-GROUP/hpx/pull/2532
https://github.com/STEllAR-GROUP/hpx/pull/2531
https://github.com/STEllAR-GROUP/hpx/pull/2529
https://github.com/STEllAR-GROUP/hpx/pull/2527
https://github.com/STEllAR-GROUP/hpx/pull/2526
• PR #2525 – Remove warnings related to unused captured variables
• Issue #2524 – CMAKE failed because it is missing: TCMALLOC_LIBRARY TCMALLOC_INCLUDE_DIR
• PR #2523 – Fixing compose_cb stack overflow
• PR #2522 – Instead of unlocking, ignore the lock while creating the message handler
• PR #2521 – Create LPROGRESS_ logging macro to simplify progress tracking and timings
• PR #2520 – Intel 17 support
• PR #2519 – Fix components example
• PR #2518 – Fixing parcel scheduling
• Issue #2517 – Race condition during Parcel Coalescing Handler creation
• Issue #2516 – HPX locks up when using at least 256 localities
• Issue #2515 – error: Install cannot find “/lib/hpx/libparcel_coalescing.so.0.9.99” but I can see that file
• PR #2514 – Making sure that all continuations of a shared_future are invoked in order
• PR #2513 – Fixing locks held during suspension
• PR #2512 – MPI Parcelport improvements and fixes related to the background work changes
• PR #2511 – Fixing bit-wise (zero-copy) serialization
• Issue #2509 – Linking errors in hwloc_topology
• PR #2508 – Added documentation for debugging with core files
• PR #2506 – Fixing background work invocations
• PR #2505 – Fix tuple serialization
• Issue #2504 – Ensure continuations are called in the order they have been attached
• PR #2503 – Adding serialization support for Vc v2 (datapar)
• PR #2502 – Resolve various, minor compiler warnings
• PR #2501 – Some other fixes around cuda examples

https://github.com/STEllAR-GROUP/hpx/pull/2525
https://github.com/STEllAR-GROUP/hpx/issues/2524
https://github.com/STEllAR-GROUP/hpx/pull/2523
https://github.com/STEllAR-GROUP/hpx/issues/2517
https://github.com/STEllAR-GROUP/hpx/pull/2516
https://github.com/STEllAR-GROUP/hpx/pull/2518
https://github.com/STEllAR-GROUP/hpx/issues/2515
https://github.com/STEllAR-GROUP/hpx/pull/2514
https://github.com/STEllAR-GROUP/hpx/pull/2513
https://github.com/STEllAR-GROUP/hpx/pull/2512
https://github.com/STEllAR-GROUP/hpx/pull/2511
https://github.com/STEllAR-GROUP/hpx/issues/2509
https://github.com/STEllAR-GROUP/hpx/pull/2508
https://github.com/STEllAR-GROUP/hpx/pull/2506
https://github.com/STEllAR-GROUP/hpx/pull/2505
https://github.com/STEllAR-GROUP/hpx/pull/2504
https://github.com/STEllAR-GROUP/hpx/pull/2503
https://github.com/STEllAR-GROUP/hpx/pull/2502
https://github.com/STEllAR-GROUP/hpx/pull/2501
• Issue #2500 - nvcc / cuda clang issue due to a missing -DHPX_WITH_CUDA flag
• PR #2499 - Adding support for std::array to wait_all and friends
• PR #2498 - Execute background work as HPX thread
• PR #2497 - Fixing configuration options for spinlock-deadlock detection
• PR #2496 - Accounting for different compilers in CrayKNL toolchain file
• PR #2494 - Adding component base class which ties a component instance to a given executor
• PR #2493 - Enable controlling amount of pending threads which must be available to allow thread stealing
• PR #2492 - Adding new command line option -hpexec:print-counter-reset
• PR #2491 - Resolve ambiguities when compiling with APEX
• PR #2490 - Resuming threads waiting on future with higher priority
• Issue #2489 - nvcc issue because -std=c++11 appears twice
• PR #2488 - Adding performance counters exposing the internal idle and busy-loop counters
• PR #2487 - Allowing for plain suspend to reschedule thread right away
• PR #2486 - Only flag HPX code for CUDA if HPX_WITH_CUDA is set
• PR #2485 - Making thread-queue parameters runtime-configurable
• PR #2484 - Added atomic counter for parcel-destinations
• PR #2483 - Added priority-queue lifo scheduler
• PR #2482 - Changing scheduler to steal only if more than a minimal number of tasks are available
• PR #2481 - Extending command line option -hpexec:print-counter-destination to support value 'none'
• PR #2479 - Added option to disable signal handler
• PR #2478 - Making sure the sine performance counter module gets loaded only for the corresponding example
• Issue #2477 - Breaking at a throw statement
• PR #2476 - Annotated function
- PR #2475 - Ensure that using %osthread% during logging will not throw for non-hpx threads
- PR #2474 - Remove now superficial non_direct actions from base_lco and friends
- PR #2473 - Refining support for ITTNotify
- PR #2472 - Some fixes around hpx compute
- Issue #2470 - redefinition of boost::detail::spinlock
- Issue #2469 - Dataflow performance issue
- PR #2468 - Perf docs update
- PR #2466 - Guarantee to execute remote direct actions on HPX-thread
- PR #2465 - Improve demo : Async copy and fixed device handling
- PR #2464 - Adding performance counter exposing instantaneous scheduler utilization
- PR #2463 - Downcast to future<void>
- PR #2462 - Fixed usage of ITT-Notify API with Intel Amplifier
- PR #2461 - Cublas demo
- PR #2460 - Fixing thread bindings
- PR #2459 - Make -std=c++11 nvcc flag consistent for in-build and installed versions
- Issue #2457 - Segmentation fault when registering a partitioned vector
- PR #2452 - Properly releasing global barrier for unhandled exceptions
- PR #2451 - Fixing long shutdown times
- PR #2450 - Attempting to fix initialization errors on newer platforms (Boost V1.63)
- PR #2449 - Replace BOOST_COMPILER_FENCE with an HPX version
- PR #2448 - This fixes a possible race in the migration code
- PR #2447 - Fixing dataflow et.al. for futures or future-ranges wrapped into ref()
- PR #2444 - Fix segfaults
• PR #2443 - Issue 2442
• Issue #2442 - Mismatch between #if/#endif and namespace scope brackets in this_thread_executers.hpp
• Issue #2441 - undeclared identifier BOOST_COMPILER_FENCE
• PR #2440 - Knl build
• PR #2438 - Datapar backend
• PR #2437 - Adapt algorithm parameter sequence changes from C++17
• PR #2436 - Adapt execution policy name changes from C++17
• Issue #2435 - Trunk broken, undefined reference to hpx::thread::interrupt(hpx::thread::id, bool)
• PR #2434 - More fixes to resource manager
• PR #2433 - Added versions of hpx::get_ptr taking client side representations
• PR #2432 - Warning fixes
• PR #2431 - Adding facility representing set of performance counters
• PR #2430 - Fix parallel_executor thread spawning
• PR #2429 - Fix attribute warning for gcc
• Issue #2427 - Seg fault running octo-tiger with latest HPX commit
• Issue #2426 - Bug in 9592f5c0bc29806ece0d8be73f35b6ca7e027edeb causes immediate crash in Octo-tiger
• PR #2425 - Fix nvcc errors due to constexpr specifier
• Issue #2424 - Async action on component present on hpx::find_here is executing synchronously
• PR #2423 - Fix nvcc errors due to constexpr specifier
• PR #2422 - Implementing hpx::this_thread thread data functions
• PR #2421 - Adding benchmark for wait_all
• Issue #2420 - Returning object of a component client from another component action fails
• PR #2419 - Infiniband parcelport

2.10. Releases
• Issue #2418 - gcc + nvcc fails to compile code that uses partitioned_vector
• PR #2417 - Fixing context switching
• PR #2416 - Adding fixes and workarounds to allow compilation with nvcc/msvc (VS2015up3)
• PR #2415 - Fix errors coming from hpx compute examples
• PR #2414 - Fixing msvc12
• PR #2413 - Enable cuda/nvcc or cuda/clang when using add_hpx_executable()
• PR #2412 - Fix issue in HPX_SetupTarget.cmake when cuda is used
• PR #2411 - This fixes the core compilation issues with MSVC12
• Issue #2410 - undefined reference to opal_hwloc191_hwloc_.....
• PR #2409 - Fixing locking for channel and receive_buffer
• PR #2407 - Solving #2402 and #2403
• PR #2406 - Improve guards
• PR #2405 - Enable parallel::for_each for iterators returning proxy types
• PR #2404 - Forward the explicitly given result_type in the hpx invoke
• Issue #2403 - datapar_execution + zip iterator: lambda arguments aren’t references
• Issue #2402 - datapar algorithm instantiated with wrong type #2402
• PR #2401 - Added support for imported libraries to HPX_Libraries.cmake
• PR #2400 - Use CMake policy CMP0060
• Issue #2399 - Error trying to push back vector of futures to vector
• PR #2398 - Allow config #defines to be written out to custom config/defines.hpp
• Issue #2397 - CMake generated config defines can cause tedious rebuilds category
• Issue #2396 - BOOST_ROOT paths are not used at link time
• PR #2395 - Fix target_link_libraries() issue when HPX Cuda is enabled

https://github.com/STEllAR-GROUP/hpx/issues/2418
https://github.com/STEllAR-GROUP/hpx/pull/2417
https://github.com/STEllAR-GROUP/hpx/pull/2416
https://github.com/STEllAR-GROUP/hpx/pull/2415
https://github.com/STEllAR-GROUP/hpx/pull/2414
https://github.com/STEllAR-GROUP/hpx/pull/2413
https://github.com/STEllAR-GROUP/hpx/pull/2412
https://github.com/STEllAR-GROUP/hpx/pull/2411
https://github.com/STEllAR-GROUP/hpx/issues/2410
https://github.com/STEllAR-GROUP/hpx/pull/2409
https://github.com/STEllAR-GROUP/hpx/pull/2407
https://github.com/STEllAR-GROUP/hpx/pull/2406
https://github.com/STEllAR-GROUP/hpx/pull/2405
https://github.com/STEllAR-GROUP/hpx/pull/2404
https://github.com/STEllAR-GROUP/hpx/issues/2403
https://github.com/STEllAR-GROUP/hpx/issues/2402
https://github.com/STEllAR-GROUP/hpx/pull/2401
https://github.com/STEllAR-GROUP/hpx/pull/2400
https://github.com/STEllAR-GROUP/hpx/issues/2399
https://github.com/STEllAR-GROUP/hpx/pull/2398
https://github.com/STEllAR-GROUP/hpx/issues/2397
https://github.com/STEllAR-GROUP/hpx/pull/2396
https://github.com/STEllAR-GROUP/hpx/pull/2395

Chapter 2. What’s so special about HPX?
• Issue #2394 - Template compilation error using HPX_WITH_DATAPAR_LIBFLATARRAY
• PR #2393 - Fixing lock registration for recursive mutex
• PR #2392 - Add keywords in target_link_libraries in hpx_setup_target
• PR #2391 - Clang goroutines
• Issue #2390 - Adapt execution policy name changes from C++17
• PR #2389 - Chunk allocator and pool are not used and are obsolete
• PR #2388 - Adding functionalities to datapar needed by octotiger
• PR #2387 - Fixing race condition for early parcels
• Issue #2386 - Lock registration broken for recursive_mutex
• PR #2385 - Datapar zip iterator
• PR #2384 - Fixing race condition in for_loop_reduction
• PR #2383 - Continuations
• PR #2382 - add LibFlatArray-based backend for datapar
• PR #2381 - remove unused typedef to get rid of compiler warnings
• PR #2380 - Tau cleanup
• PR #2379 - Can send immediate
• PR #2378 - Renaming copy_helper/copy_n_helper/move_helper/move_n_helper
• Issue #2376 - Boost trunk’s spinlock initializer fails to compile
• PR #2375 - Add support for minimal thread local data
• PR #2374 - Adding API functions set_config_entry_callback
• PR #2373 - Add a simple utility for debugging that gives suspended task backtraces
• PR #2372 - Barrier Fixes
• Issue #2370 - Can’t wait on a wrapped future

https://github.com/STEllAR-GROUP/hpx/issues/2394
https://github.com/STEllAR-GROUP/hpx/pull/2393
https://github.com/STEllAR-GROUP/hpx/pull/2392
https://github.com/STEllAR-GROUP/hpx/pull/2391
https://github.com/STEllAR-GROUP/hpx/issues/2390
https://github.com/STEllAR-GROUP/hpx/pull/2389
https://github.com/STEllAR-GROUP/hpx/pull/2388
https://github.com/STEllAR-GROUP/hpx/pull/2387
https://github.com/STEllAR-GROUP/hpx/issue/2386
https://github.com/STEllAR-GROUP/hpx/pull/2385
https://github.com/STEllAR-GROUP/hpx/pull/2384
https://github.com/STEllAR-GROUP/hpx/pull/2383
https://github.com/STEllAR-GROUP/hpx/pull/2382
https://github.com/STEllAR-GROUP/hpx/pull/2381
https://github.com/STEllAR-GROUP/hpx/pull/2380
https://github.com/STEllAR-GROUP/hpx/pull/2379
https://github.com/STEllAR-GROUP/hpx/pull/2378
https://github.com/STEllAR-GROUP/hpx/issues/2376
https://github.com/STEllAR-GROUP/hpx/pull/2375
https://github.com/STEllAR-GROUP/hpx/pull/2374
https://github.com/STEllAR-GROUP/hpx/pull/2373
https://github.com/STEllAR-GROUP/hpx/pull/2372
https://github.com/STEllAR-GROUP/hpx/issue/2370
• PR #2369 - Fixing stable_partition
• PR #2367 - Fixing find_prefixes for Windows platforms
• PR #2366 - Testing for experimental/optional only in C++14 mode
• PR #2364 - Adding set_config_entry
• PR #2363 - Fix papi
• PR #2362 - Adding missing macros for new non-direct actions
• PR #2361 - Improve cmake output to help debug compiler incompatibility check
• PR #2360 - Fixing race condition in condition_variable
• PR #2359 - Fixing shutdown when parcels are still in flight
• Issue #2357 - Failed to insert console_print_action into typename_to_id_t registry
• PR #2356 - Fixing return type of get_iterator_tuple
• PR #2355 - Fixing compilation against Boost 1.62
• PR #2354 - Adding serialization for mask_type if CPU_COUNT > 64
• PR #2353 - Adding hooks to tie in APEX into the parcel layer
• Issue #2352 - Compile errors when using intel 17 beta (for KNL) on edison
• PR #2351 - Fix function vtable get_function_address implementation
• Issue #2350 - Build failure - master branch (4de09f5) with Intel Compiler v17
• PR #2349 - Enabling zero-copy serialization support for std::vector
• PR #2348 - Adding test to verify #2334 is fixed
• PR #2347 - Bug fixes for hpx.compute and hpx::lcos::channel
• PR #2346 - Removing cmake “find” files that are in the APEX cmake Modules
• PR #2345 - Implemented parallel::stable_partition
• PR #2344 - Making hpx::lcos::channel usable with basename registration
• PR #2343 - Fix a couple of examples that failed to compile after recent api changes
• Issue #2342 - Enabling APEX causes link errors
• PR #2341 - Removing cmake “find” files that are in the APEX cmake Modules
• PR #2340 - Implemented all existing datapar algorithms using Boost.SIMD
• PR #2339 - Fixing 2338
• PR #2338 - Possible race in sliding semaphore
• PR #2337 - Adjust osu_latency test to measure window_size parcels in flight at once
• PR #2336 - Allowing remote direct actions to be executed without spawning a task
• PR #2335 - Making sure multiple components are properly initialized from arguments
• Issue #2334 - Cannot construct component with large vector on a remote locality
• PR #2332 - Fixing hpx::lcos::local::barrier
• PR #2331 - Updating APEX support to include OTF2
• PR #2330 - Support for data-parallelism for parallel algorithms
• Issue #2329 - Coordinate settings in cmake
• PR #2328 - fix LibGeoDecomp builds with HPX + GCC 5.3.0 + CUDA 8RC
• PR #2326 - Making scan_partitioner work (for now)
• Issue #2323 - Constructing a vector of components only correctly initializes the first component
• PR #2322 - Fix problems that bubbled up after merging #2278
• PR #2321 - Scalable barrier
• PR #2320 - Std flag fixes
• Issue #2319 - -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 - Improve handling of –hpx:bind=<bind-spec>
• PR #2317 - Making sure command line warnings are printed once only

2.10. Releases
• PR #2316 - Fixing command line handling for default bind mode
• PR #2315 - Set id_retrieved if set_id is present
• Issue #2314 - Warning for requested/allocated thread discrepancy is printed twice
• Issue #2313 - --hpx:print-bind doesn’t work with --hpx:pu-step
• Issue #2312 - --hpx:bind range specifier restrictions are overly restrictive
• Issue #2311 - hpx_0.9.99 out of project build fails
• PR #2310 - Simplify function registration
• PR #2309 - Spelling and grammar revisions in documentation (and some code)
• PR #2306 - Correct minor typo in the documentation
• PR #2305 - Cleaning up and fixing parcel coalescing
• PR #2304 - Inspect checks for stream related includes
• PR #2303 - Add functionality allowing to enumerate threads of given state
• PR #2301 - Algorithm overloads fix for VS2013
• PR #2300 - Use <cstdint>, add inspect checks
• PR #2299 - Replace boost::[c]ref with std::[c]ref, add inspect checks
• PR #2297 - Fixing compilation with no hw_loc
• PR #2296 - Hpx compute
• PR #2293 - Making sure for_loop(execution::par, 0, N, . . . ) is actually executed in parallel
• PR #2294 - Throwing exceptions if the runtime is not up and running
• PR #2293 - Removing unused parcel port code
• PR #2292 - Refactor function vtables
• PR #2291 - Fixing 2286
• PR #2290 - Simplify algorithm overloads
- PR #2289 - Adding performance counters reporting parcel related data on a per-action basis
- Issue #2288 - Remove dormant parcelports
- Issue #2286 - adjustments to parcel handling to support parcelports that do not need a connection cache
- PR #2285 - add CMake option to disable package export
- PR #2283 - Add more inspect checks for use of deprecated components
- Issue #2282 - Arithmetic exception in executor static chunker
- Issue #2281 - For loop doesn’t parallelize
- PR #2280 - Fixing 2277: build failure with PAPI
- PR #2279 - Child vs parent stealing
- Issue #2277 - master branch build failure (53c5b4f) with papi
- PR #2276 - Compile time launch policies
- PR #2275 - Replace boost::chrono with std::chrono in interfaces
- PR #2274 - Replace most uses of Boost.Assign with initializer list
- PR #2273 - Fixed typos
- PR #2272 - Inspect checks
- PR #2271 - Adding test verifying -lhpox.os_threads=all
- PR #2269 - Added inspect check for now obsolete boost type traits
- PR #2268 - Moving more code into source files
- Issue #2267 - Add inspect support to deprecate Boost.TypeTraits
- PR #2266 - Adding channel LCO
- PR #2264 - Make support for std::ref mandatory
- PR #2263 - Constrain tuple_member forwarding constructor
- Issue #2262 - Test hpx.os_threads=all

See the [GitHub](https://github.com/STEllAR-GROUP/hpx/pull/2289) and [GitHub](https://github.com/STEllAR-GROUP/hpx/issues/2288) pages for more details.
• Issue #2261 - OS X: Error: no matching constructor for initialization of ‘hpx::lcos::local::condition_variable_any’
• Issue #2260 - Make support for std::ref mandatory
• PR #2259 - Remove most of Boost.MPL, Boost.EnableIf and Boost.TypeTraits
• PR #2258 - Fixing #2256
• PR #2257 - Fixing launch process
• Issue #2256 - Actions are not registered if not invoked
• PR #2255 - Coalescing histogram
• PR #2254 - Silence explicit initialization in copy-constructor warnings
• PR #2253 - Drop support for GCC 4.6 and 4.7
• PR #2252 - Prepare V1.0
• PR #2251 - Convert to 0.9.99
• PR #2249 - Adding iterator_facade and iterator_adaptor
• Issue #2248 - Need a feature to yield to a new task immediately
• PR #2247 - Adding split_future
• PR #2245 - Add an example for handing over a component instance to a dynamically launched locality
• Issue #2243 - Add example demonstrating AGAS symbolic name registration
• Issue #2242 - pkgconfig test broken on CentOS 7 / Boost 1.61
• Issue #2241 - Compilation error for partitioned vector in hpx_compute branch
• PR #2240 - Fixing termination detection on one locality
• Issue #2239 - Create a new facility lcos::split_all
• Issue #2236 - hpx::cout vs. std::cout
• PR #2234 - Implement local-only primary namespace service
• Issue #2147 - would like to know how much data is being routed by particular actions
• Issue #2109 - Warning while compiling hpx
• Issue #1973 - Setting INTERFACE_COMPILE_OPTIONS for hpx_init in CMake taints Fortran_FLAGS
• Issue #1864 - run_guarded using bound function ignores reference
• Issue #1754 - Running with TCP parcelport causes immediate crash or freeze
• Issue #1655 - Enable zip_iterator to be used with Boost traversal iterator categories
• Issue #1591 - Optimize AGAS for shared memory only operation
• Issue #1401 - Need an efficient infiniband parcelport
• Issue #1125 - Fix the IPC parcelport
• Issue #839 - Refactor ibverbs and shmem parcelport
• Issue #702 - Add instrumentation of parcel layer
• Issue #668 - Implement ispc task interface
• Issue #533 - Thread queue/deque internal parameters should be runtime configurable
• Issue #475 - Create a means of combining performance counters into querysets

2.10.13 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.

https://github.com/STEllAR-GROUP/hpx/issues/2109
https://github.com/STEllAR-GROUP/hpx/issues/1973
https://github.com/STEllAR-GROUP/hpx/issues/1864
https://github.com/STEllAR-GROUP/hpx/issues/1754
https://github.com/STEllAR-GROUP/hpx/issues/1655
https://github.com/STEllAR-GROUP/hpx/issues/1591
https://github.com/STEllAR-GROUP/hpx/issues/1401
https://github.com/STEllAR-GROUP/hpx/issues/1125
https://github.com/STEllAR-GROUP/hpx/issues/839
https://github.com/STEllAR-GROUP/hpx/issues/702
https://github.com/STEllAR-GROUP/hpx/issues/668
https://github.com/STEllAR-GROUP/hpx/issues/533
https://github.com/STEllAR-GROUP/hpx/issues/475
• We have started to add parallel algorithm overloads based on the C++ Extensions for Ranges (N4560) proposal. This also includes the addition of projections to the existing algorithms. Please see Issue #1668 for a list of algorithms which have been adapted to N4560.

• We have implemented index-based parallel for-loops based on a corresponding standardization proposal (P0075R1). Please see Issue #2016 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 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.

• 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 and PR #1966 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 and PR #1912 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

---

https://github.com/STEllAR-GROUP/hpx/issues/1668
https://github.com/STEllAR-GROUP/hpx/issues/2016
https://github.com/STEllAR-GROUP/hpx/issues/1141
https://github.com/STEllAR-GROUP/hpx/issues/559
https://github.com/STEllAR-GROUP/hpx/pull/1966
https://github.com/STEllAR-GROUP/hpx/pull/1912
https://github.com/STEllAR-GROUP/hpx/pull/1966
https://github.com/STEllAR-GROUP/hpx/pull/1912
https://github.com/STEllAR-GROUP/hpx/pull/1966
https://github.com/STEllAR-GROUP/hpx/pull/1912
https://github.com/STEllAR-GROUP/hpx/pull/1966
https://github.com/STEllAR-GROUP/hpx/pull/1912
https://github.com/STEllAR-GROUP/hpx/pull/1966
https://github.com/STEllAR-GROUP/hpx/pull/1912
(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\(^{3086}\)) 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`\(^{3087}\) 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\(^{3088}\)) proposal imposes some breaking changes related to the return types of some of the parallel algorithms. Please see Issue #1668\(^{3089}\) 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_policy` 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).

\(^{3086}\) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf

\(^{3087}\) https://www.cmake.org

\(^{3088}\) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf

\(^{3089}\) https://github.com/STEllAR-GROUP/hpx/issues/1668
Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2250 - change default chunker of parallel executor to static one
- PR #2247 - HPX on ppc64le
- PR #2244 - Fixing MSVC problems
- PR #2238 - Fixing small typos
- PR #2237 - Fixing small typos
- PR #2234 - Fix broken add test macro when extra args are passed in
- PR #2231 - Fixing possible race during future awaiting in serialization
- PR #2230 - Fix stream nvcc
- PR #2229 - Fixed run_as_hpx_thread
- PR #2228 - On prefectching_test branch : adding prefetching_iterator and related tests used for prefetching containers within lambda functions
- PR #2227 - Support for HPXCL’s opencl::event
- PR #2226 - Preparing for release of V0.9.99
- PR #2225 - fix issue when compiling components with hpxcxx
- PR #2224 - Compute alloc fix
- PR #2223 - Simplify promise
- PR #2222 - Replace last uses of boost::function by util::function_nonser
- PR #2221 - Fix config tests
- PR #2220 - Fixing gcc 4.6 compilation issues
- PR #2219 - nullptr support for [unique_]function
- PR #2218 - Introducing clang tidy
- PR #2216 - Replace NULL with nullptr
• Issue #2214\textsuperscript{3111} - Let inspect flag use of NULL, suggest nullptr instead
• PR #2213\textsuperscript{3112} - Require support for nullptr
• PR #2212\textsuperscript{3113} - Properly find jemalloc through pkg-config
• PR #2211\textsuperscript{3114} - Disable a couple of warnings reported by Intel on Windows
• PR #2210\textsuperscript{3115} - Fixed host::block_allocator::bulk_construct
• PR #2209\textsuperscript{3116} - Started to clean up new sort algorithms, made things compile for sort_by_key
• PR #2208\textsuperscript{3117} - A couple of fixes that were exposed by a new sort algorithm
• PR #2207\textsuperscript{3118} - Adding missing includes in /hpx/include/serialization.hpp
• PR #2206\textsuperscript{3119} - Call package_action::get_future before package_action::apply
• PR #2205\textsuperscript{3120} - The indirect_packaged_task::operator() needs to be run on a HPX thread
• PR #2204\textsuperscript{3121} - Variadic executor parameters
• PR #2203\textsuperscript{3122} - Delay-initialize members of partitioned iterator
• PR #2202\textsuperscript{3123} - Added segmented fill for hpx::vector
• Issue #2201\textsuperscript{3124} - Null Thread id encountered on partitioned_vector
• PR #2200\textsuperscript{3125} - Fix hangs
• PR #2199\textsuperscript{3126} - Deprecating hpx/traits.hpp
• PR #2198\textsuperscript{3127} - Making explicit inclusion of external libraries into build
• PR #2197\textsuperscript{3128} - Fix typo in QT CMakeLists
• PR #2196\textsuperscript{3129} - Fixing a gcc warning about attributes being ignored
• PR #2194\textsuperscript{3130} - Fixing partitioned_vector_spmd_foreach example
• Issue #2193\textsuperscript{3131} - partitioned_vector_spmd_foreach seg faults
• PR #2192\textsuperscript{3132} - Support Boost.Thread v4
• PR #2191\textsuperscript{3133} - HPX.Compute prototype

\textsuperscript{3111} https://github.com/STEllAR-GROUP/hpx/issues/2214
\textsuperscript{3112} https://github.com/STEllAR-GROUP/hpx/pull/2213
\textsuperscript{3113} https://github.com/STEllAR-GROUP/hpx/pull/2212
\textsuperscript{3114} https://github.com/STEllAR-GROUP/hpx/pull/2211
\textsuperscript{3115} https://github.com/STEllAR-GROUP/hpx/pull/2210
\textsuperscript{3116} https://github.com/STEllAR-GROUP/hpx/pull/2209
\textsuperscript{3117} https://github.com/STEllAR-GROUP/hpx/pull/2208
\textsuperscript{3118} https://github.com/STEllAR-GROUP/hpx/pull/2207
\textsuperscript{3119} https://github.com/STEllAR-GROUP/hpx/pull/2206
\textsuperscript{3120} https://github.com/STEllAR-GROUP/hpx/pull/2205
\textsuperscript{3121} https://github.com/STEllAR-GROUP/hpx/pull/2204
\textsuperscript{3122} https://github.com/STEllAR-GROUP/hpx/pull/2203
\textsuperscript{3123} https://github.com/STEllAR-GROUP/hpx/pull/2202
\textsuperscript{3124} https://github.com/STEllAR-GROUP/hpx/issues/2201
\textsuperscript{3125} https://github.com/STEllAR-GROUP/hpx/pull/2200
\textsuperscript{3126} https://github.com/STEllAR-GROUP/hpx/pull/2199
\textsuperscript{3127} https://github.com/STEllAR-GROUP/hpx/pull/2198
\textsuperscript{3128} https://github.com/STEllAR-GROUP/hpx/pull/2197
\textsuperscript{3129} https://github.com/STEllAR-GROUP/hpx/pull/2196
\textsuperscript{3130} https://github.com/STEllAR-GROUP/hpx/pull/2194
\textsuperscript{3131} https://github.com/STEllAR-GROUP/hpx/issues/2193
\textsuperscript{3132} https://github.com/STEllAR-GROUP/hpx/pull/2192
\textsuperscript{3133} https://github.com/STEllAR-GROUP/hpx/pull/2191
• PR #2190 - Spawning operation on new thread if remaining stack space becomes too small
• PR #2189 - Adding callback taking index and future to when_each
• PR #2188 - Adding new example demonstrating receive_buffer
• PR #2187 - Mask 128-bit ints if CUDA is being used
• PR #2186 - Make startup & shutdown functions unique_function
• PR #2185 - Fixing logging output not to cause hang on shutdown
• PR #2184 - Allowing component clients as action return types
• Issue #2183 - Enabling logging output causes hang on shutdown
• Issue #2182 - 1d_stencil seg fault
• Issue #2181 - Setting small stack size does not change default
• PR #2180 - Changing default bind mode to balanced
• PR #2179 - adding prefetching_iterator and related tests used for prefetching containers within lambda functions
• PR #2177 - Fixing 2176
• Issue #2176 - Launch process test fails on OSX
• PR #2175 - Fix unbalanced config/warnings includes, add some new ones
• PR #2174 - Fix test categorization : regression not unit
• Issue #2172 - Different performance results
• Issue #2171 - “negative entry in reference count table” running octotiger on 32 nodes on queenbee
• Issue #2170 - Error while compiling on Mac + boost 1.60
• PR #2168 - Fixing problems with is_bitwise_serializable
• Issue #2167 - startup & shutdown function should accept unique_function
• Issue #2166 - Simple receive_buffer example
• PR #2165 - Fix wait all
• PR #2164 - Fix wait all
• PR #2163 - Fix some typos in config tests
• PR #2162 - Improve #includes
• PR #2160 - Add inspect check for missing #include <list>
• PR #2159 - Add missing finalize call to stop test hanging
• PR #2158 - Algo fixes
• PR #2157 - Stack check
• Issue #2156 - OSX reports stack space incorrectly (generic context coroutines)
• Issue #2155 - Race condition suspected in runtime
• PR #2154 - Replace boost::detail::atomic_count with the new util::atomic_count
• PR #2153 - Fix stack overflow on OSX
• PR #2152 - Define is_bitwise_serializable as is_trivially_copyable when available
• PR #2151 - Adding missing <cstring> for std::mem* functions
• Issue #2150 - Unable to use component clients as action return types
• PR #2149 - std::memmove copies bytes, use bytes*sizeof(type) when copying larger types
• PR #2148 - Adding customization point for parallel copy/move
• PR #2147 - Applying changes to address warnings issued by latest version of PVS Studio
• Issue #2146 - hpx::parallel::copy is broken after trivially copyable changes
• PR #2145 - Some minor tweaks to compute prototype
• PR #2144 - Added Boost version support information over OSX platform
• PR #2143 - Fixing memory leak in example
• PR #2142 - Add missing specializations in execution policies
• PR #2141 - This PR fixes a few problems reported by Clang’s Undefined Behavior sanitizer

3157 https://github.com/STEllAR-GROUP/hpx/pull/2164
3158 https://github.com/STEllAR-GROUP/hpx/pull/2163
3159 https://github.com/STEllAR-GROUP/hpx/pull/2162
3160 https://github.com/STEllAR-GROUP/hpx/pull/2160
3161 https://github.com/STEllAR-GROUP/hpx/pull/2159
3162 https://github.com/STEllAR-GROUP/hpx/pull/2158
3163 https://github.com/STEllAR-GROUP/hpx/pull/2157
3164 https://github.com/STEllAR-GROUP/hpx/issues/2156
3165 https://github.com/STEllAR-GROUP/hpx/issues/2155
3166 https://github.com/STEllAR-GROUP/hpx/pull/2154
3167 https://github.com/STEllAR-GROUP/hpx/pull/2153
3168 https://github.com/STEllAR-GROUP/hpx/pull/2152
3169 https://github.com/STEllAR-GROUP/hpx/pull/2151
3170 https://github.com/STEllAR-GROUP/hpx/issues/2150
3171 https://github.com/STEllAR-GROUP/hpx/pull/2149
3172 https://github.com/STEllAR-GROUP/hpx/pull/2146
3173 https://github.com/STEllAR-GROUP/hpx/pull/2145
3174 https://github.com/STEllAR-GROUP/hpx/pull/2144
3175 https://github.com/STEllAR-GROUP/hpx/pull/2143
3176 https://github.com/STEllAR-GROUP/hpx/pull/2142
3177 https://github.com/STEllAR-GROUP/hpx/pull/2141
3178 https://github.com/STEllAR-GROUP/hpx/pull/2139
• PR #2138 - Revert “Adding fedora docs”
• PR #2136 - Removed double semicolon
• PR #2135 - Add deprecated #include check for hpx_fwd.hpp
• PR #2134 - Resolved memory leak in stencil_8
• PR #2133 - Replace uses of boost pointer containers
• PR #2132 - Removing unused typedef
• PR #2131 - Add several include checks for std facilities
• PR #2130 - Fixing parcel compression, adding test
• PR #2129 - Fix invalid attribute warnings
• Issue #2128 - hpx::init seems to segfault
• PR #2127 - Making executor_traits N-ary
• PR #2126 - GCC 4.6 fails to deduce the correct type in lambda
• PR #2125 - Making parcel coalescing test actually test something
• Issue #2124 - Make a testcase for parcel compression
• Issue #2123 - hpx/hpx/runtime/applier_fwd.hpp - Multiple defined types
• Issue #2122 - Exception in primary_namespace::resolve_free_list
• Issue #2121 - Possible memory leak in 1d_stencil_8
• PR #2120 - Fixing 2119
• Issue #2119 - reduce_by_key compilation problems
• Issue #2118 - Premature unwrapping of boost::ref’ed arguments
• PR #2117 - Added missing initializer on last constructor for thread_description
• PR #2116 - Use a lightweight bind implementation when no placeholders are given
• PR #2115 - Replace boost::shared_ptr with std::shared_ptr

https://github.com/STEllAR-GROUP/hpx/pull/2138
https://github.com/STEllAR-GROUP/hpx/pull/2136
https://github.com/STEllAR-GROUP/hpx/pull/2135
https://github.com/STEllAR-GROUP/hpx/pull/2134
https://github.com/STEllAR-GROUP/hpx/pull/2133
https://github.com/STEllAR-GROUP/hpx/pull/2132
https://github.com/STEllAR-GROUP/hpx/pull/2131
https://github.com/STEllAR-GROUP/hpx/pull/2130
https://github.com/STEllAR-GROUP/hpx/pull/2129
https://github.com/STEllAR-GROUP/hpx/issues/2128
https://github.com/STEllAR-GROUP/hpx/issues/2127
https://github.com/STEllAR-GROUP/hpx/issues/2126
https://github.com/STEllAR-GROUP/hpx/issues/2125
https://github.com/STEllAR-GROUP/hpx/issues/2124
https://github.com/STEllAR-GROUP/hpx/issues/2123
https://github.com/STEllAR-GROUP/hpx/issues/2122
https://github.com/STEllAR-GROUP/hpx/issues/2121
https://github.com/STEllAR-GROUP/hpx/issues/2119
https://github.com/STEllAR-GROUP/hpx/issues/2118
https://github.com/STEllAR-GROUP/hpx/pull/2117
https://github.com/STEllAR-GROUP/hpx/pull/2116
https://github.com/STEllAR-GROUP/hpx/pull/2115
• PR #2114 - Adding hook functions for executor_parameter_traits supporting timers
• Issue #2113 - Compilation error with gcc version 4.9.3 (MacPorts gcc49 4.9.3_0)
• PR #2112 - Replace uses of safe_bool with explicit operator bool
• Issue #2111 - Compilation error on QT example
• Issue #2110 - Compilation error when passing non-future argument to unwrapped continuation in dataflow
• Issue #2109 - Warning while compiling hpx
• Issue #2109 - Stack trace of last bug causing issues with octotiger
• Issue #2108 - Stack trace of last bug causing issues with octotiger
• PR #2107 - Making sure that a missing parcel_coalescing module does not cause startup exceptions
• PR #2106 - Stop using hpx_fwd.hpp
• Issue #2105 - coalescing plugin handler is not optional any more
• Issue #2104 - Make executor_traits N-nary
• Issue #2103 - Build error with octotiger and hpx commit e657426d
• PR #2102 - Combining thread data storage
• PR #2101 - Added repartition version of 1d stencil that uses any performance counter
• PR #2100 - Drop obsolete TR1 result_of protocol
• PR #2099 - Replace uses of boost::bind with util::bind
• PR #2098 - Deprecated inspect checks
• PR #2097 - Reduce by key, extends #1141
• PR #2096 - Moving local cache from external to hpx/util
• PR #2095 - Bump minimum required Boost to 1.50.0
• PR #2094 - Add include checks for several Boost utilities
• Issue #2093 - /.../local_cache.hpp(89): error #303: explicit type is missing (“int” assumed)
- PR #2091 - Fix for Raspberry pi build
- PR #2090 - Fix storage size for util::function<> 
- PR #2089 - Fix #2088
- Issue #2088 - More verbose output from cmake configuration
- PR #2087 - Making sure init_globally always executes hpx_main
- Issue #2086 - Race condition with recent HPX
- PR #2085 - Adding #include checker
- PR #2084 - Replace boost lock types with standard library ones
- PR #2083 - Simplify packaged task
- PR #2082 - Updating APEX version for testing
- PR #2081 - Cleanup exception headers
- PR #2080 - Make call_once variadic
- Issue #2079 - With GNU C++, line 85 of hpx/config/version.hpp causes link failure when linking application
- Issue #2078 - Simple test fails with _GLIBCXX_DEBUG defined
- PR #2077 - Instantiate board in nqueen client
- PR #2076 - Moving coalescing registration to TUs
- PR #2075 - Fixed some documentation typos
- PR #2074 - Adding flush-mode to message handler flush
- PR #2073 - Fixing performance regression introduced lately
- PR #2072 - Refactor local::condition_variable
- PR #2071 - Timer based on boost::asio::deadline_timer
- PR #2070 - Refactor tuple based functionality
- PR #2069 - Fixed typos
• Issue #20683249 - Seg fault with octotiger
• PR #20673250 - Algorithm cleanup
• PR #20663251 - Split credit fixes
• PR #20653252 - Rename HPX_MOVABLE_BUT_NOT_COPYABLE to HPX_MOVABLE_ONLY
• PR #20643253 - Fixed some typos in docs
• PR #20633254 - Adding example demonstrating template components
• Issue #20623255 - Support component templates
• PR #20613256 - Replace some uses of lexical_cast<string> with C++11 std::to_string
• PR #20603257 - Replace uses of boost::noncopyable with HPX_NON_COPYABLE
• PR #20593258 - Adding missing for_loop algorithms
• PR #20583259 - Move several definitions to more appropriate headers
• PR #20573260 - Simplify assert_owns_lock and ignore_while_checking
• PR #20563261 - Replacing std::result_of with util::result_of
• PR #20553262 - Fix process launching/connecting back
• PR #20543263 - Add a forwarding coroutine header
• PR #20533264 - Replace uses of boost::unordered_map with std::unordered_map
• PR #20523265 - Rewrite tuple unwrap
• PR #20513266 - Replace uses of BOOST_SCOPED_ENUM with C++11 scoped enums
• PR #20493267 - Attempt to narrow down split_credit problem
• PR #20483268 - Fixing gcc startup hangs
• PR #20473269 - Fixing when_xxx and wait_xxx for MSVC12
• PR #20463270 - adding persistent_auto_chunk_size and related tests for for_each
• PR #20453271 - Fixing HPX_HAVE_THREAD_BACKTRACE_DEPTH build time configuration

3249 https://github.com/STEllAR-GROUP/hpx/issues/2068
3250 https://github.com/STEllAR-GROUP/hpx/pull/2067
3251 https://github.com/STEllAR-GROUP/hpx/pull/2066
3252 https://github.com/STEllAR-GROUP/hpx/pull/2065
3253 https://github.com/STEllAR-GROUP/hpx/pull/2064
3254 https://github.com/STEllAR-GROUP/hpx/pull/2063
3255 https://github.com/STEllAR-GROUP/hpx/issues/2062
3256 https://github.com/STEllAR-GROUP/hpx/pull/2061
3257 https://github.com/STEllAR-GROUP/hpx/pull/2060
3258 https://github.com/STEllAR-GROUP/hpx/pull/2059
3259 https://github.com/STEllAR-GROUP/hpx/pull/2058
3260 https://github.com/STEllAR-GROUP/hpx/pull/2057
3261 https://github.com/STEllAR-GROUP/hpx/pull/2056
3262 https://github.com/STEllAR-GROUP/hpx/pull/2055
3263 https://github.com/STEllAR-GROUP/hpx/pull/2054
3264 https://github.com/STEllAR-GROUP/hpx/pull/2053
3265 https://github.com/STEllAR-GROUP/hpx/pull/2052
3266 https://github.com/STEllAR-GROUP/hpx/pull/2050
3267 https://github.com/STEllAR-GROUP/hpx/pull/2049
3268 https://github.com/STEllAR-GROUP/hpx/pull/2048
3269 https://github.com/STEllAR-GROUP/hpx/pull/2047
3270 https://github.com/STEllAR-GROUP/hpx/pull/2046
3271 https://github.com/STEllAR-GROUP/hpx/pull/2045

2.10. Releases
• PR #2044 - Adding missing service executor types
• PR #2043 - Removing ambiguous definitions for is_future_range and future_range_traits
• PR #2042 - Clarify that HPX builds can use (much) more than 2GB per process
• PR #2041 - Changing future_iterator_traits to support pointers
• Issue #2040 - Improve documentation memory usage warning?
• PR #2039 - Coroutine cleanup
• PR #2038 - Fix cmake policy CMP0042 warning MACOSX_RPATH
• PR #2037 - Avoid redundant specialization of [unique]function_nonser
• PR #2036 - nvcc dies with an internal error upon pushing/popping warnings inside templates
• Issue #2035 - Use a less restrictive iterator definition in hpx::lcos::detail::future_iterator_traits
• PR #2034 - Fixing compilation error with thread queue wait time performance counter
• Issue #2033 - Compilation error when compiling with thread queue waittime performance counter
• Issue #2032 - Ambiguous template instantiation for is_future_range and future_range_traits.
• PR #2031 - Don’t restart timer on every incoming parcel
• PR #2030 - Unify handling of execution policies in parallel algorithms
• PR #2029 - Make pkg-config .pc files use .dylib on OSX
• PR #2028 - Adding process component
• PR #2027 - Making check for compiler compatibility independent on compiler path
• PR #2025 - Fixing inspect tool
• PR #2024 - Intel13 removal
• PR #2023 - Fix errors related to older boost versions and parameter pack expansions in lambdas
• Issue #2022 - gmake fail: “No rule to make target /usr/lib46/libboost_context-mt.so”
• PR #2021 - Added Sudoku example
• Issue #2020 - Make errors related to init_globally.cpp example while building HPX out of the box
• PR #2019 - Fixed some compilation and cmake errors encountered in nqueen example
• PR #2018 - For loop algorithms
• PR #2017 - Non-recursive at_index implementation
• Issue #2016 - Add index-based for-loops
• Issue #2015 - Change default bind-mode to balanced
• PR #2014 - Fixed dataflow if invoked action returns a future
• PR #2013 - Fixing compilation issues with external example
• PR #2012 - Added Sierpinski Triangle example
• Issue #2011 - Compilation error while running sample hello_world_component code
• PR #2010 - Segmented move implemented for hpx::vector
• Issue #2009 - pkg-config order incorrect on 14.04 / GCC 4.8
• Issue #2008 - Compilation error in dataflow of action returning a future
• PR #2007 - Adding new performance counter exposing overall scheduler time
• PR #2006 - Function includes
• PR #2005 - Adding an example demonstrating how to initialize HPX from a global object
• PR #2004 - Fixing 2000
• PR #2003 - Adding generation parameter to gather to enable using it more than once
• PR #2002 - Turn on position independent code to solve link problem with hpx_init
• Issue #2001 - Gathering more than once segfaults
• Issue #2000 - Undefined reference to hpx::assertion_failed
• Issue #1999 - Seg fault in hpx::lcos::base_lco_with_value<*>::set_value_nonvirt() when running octo-tiger
• PR #1998 - Detect unknown command line options

https://github.com/STEllAR-GROUP/hpx/issues/2020
https://github.com/STEllAR-GROUP/hpx/pull/2019
https://github.com/STEllAR-GROUP/hpx/pull/2018
https://github.com/STEllAR-GROUP/hpx/pull/2017
https://github.com/STEllAR-GROUP/hpx/pull/2016
https://github.com/STEllAR-GROUP/hpx/issues/2015
https://github.com/STEllAR-GROUP/hpx/pull/2014
https://github.com/STEllAR-GROUP/hpx/pull/2013
https://github.com/STEllAR-GROUP/hpx/pull/2012
https://github.com/STEllAR-GROUP/hpx/pull/2011
https://github.com/STEllAR-GROUP/hpx/pull/2010
https://github.com/STEllAR-GROUP/hpx/issues/2008
https://github.com/STEllAR-GROUP/hpx/pull/2005
https://github.com/STEllAR-GROUP/hpx/pull/2004
https://github.com/STEllAR-GROUP/hpx/pull/2003
https://github.com/STEllAR-GROUP/hpx/pull/2002
https://github.com/STEllAR-GROUP/hpx/pull/2001
https://github.com/STEllAR-GROUP/hpx/pull/2000
https://github.com/STEllAR-GROUP/hpx/issues/1999
• PR #1997 - Extending thread description
• PR #1996 - Adding natvis files to solution (MSVC only)
• Issue #1995 - Command line handling does not produce error
• PR #1994 - Possible missing include in test_utils.hpp
• PR #1993 - Add missing LANGUAGES tag to a hpx_add_compile_flag_if_available() call in CMake-Lists.txt
• PR #1992 - Fixing shared_executor_test
• PR #1991 - Making sure the winsock library is properly initialized
• PR #1990 - Fixing bind_test placeholder ambiguity coming from boost-1.60
• PR #1989 - Possible missing include in test_utils.hpp
• PR #1988 - Performance tuning
• PR #1987 - Make configurable size of internal storage in util::function
• PR #1986 - AGAS Refactoring+1753 Cache mods
• PR #1985 - Adding missing task_block::run() overload taking an executor
• PR #1984 - Adding an optimized LRU Cache implementation (for AGAS)
• PR #1983 - Avoid invoking migration table look up for all objects
• PR #1981 - Replacing uintptr_t (which is not defined everywhere) with std::size_t
• PR #1980 - Optimizing LCO continuations
• PR #1979 - Fixing Cori
• PR #1978 - Fix test check that got broken in hasty fix to memory overflow
• PR #1977 - Refactor action traits
• PR #1976 - Fixes typo in README.rst
• PR #1975 - Reduce size of benchmark timing arrays to fix test failures
• PR #1974 - Add action to update data owned by the partitioned_vector component
• PR #1972 - Adding partitioned_vector SPMD example
- PR #1971 - Fixing 1965
- PR #1970 - Papi fixes
- PR #1969 - Fixing continuation recursions to not depend on fixed amount of recursions
- PR #1968 - More segmented algorithms
- Issue #1967 - Simplify component implementations
- PR #1966 - Migrate components
- Issue #1964 - fatal error: ‘boost/lockfree/detail/branch_hints.hpp’ file not found
- Issue #1962 - parallel:copy_if has race condition when used on in place arrays
- PR #1963 - Fixing Static Parcelport initialization
- PR #1961 - Fix function target
- Issue #1960 - Papi counters don’t reset
- PR #1959 - Fixing 1958
- Issue #1958 - inclusive_scan gives incorrect results with non-commutative operator
- PR #1957 - Fixing #1950
- PR #1956 - Sort by key example
- PR #1955 - Adding regression test for #1946: Hang in wait_all() in distributed run
- Issue #1954 - HPX releases should not use -Werror
- PR #1953 - Adding performance analysis for AGAS cache
- PR #1952 - Adapting test for explicit variadics to fail for gcc 4.6
- PR #1951 - Fixing memory leak
- Issue #1950 - Simplify external builds
- PR #1949 - Fixing yet another lock that is being held during suspension
- PR #1948 - Fixed container algorithms for Intel

https://github.com/STEllAR-GROUP/hpx/pull/1971
https://github.com/STEllAR-GROUP/hpx/pull/1970
https://github.com/STEllAR-GROUP/hpx/pull/1969
https://github.com/STEllAR-GROUP/hpx/pull/1968
https://github.com/STEllAR-GROUP/hpx/issues/1967
https://github.com/STEllAR-GROUP/hpx/pull/1966
https://github.com/STEllAR-GROUP/hpx/issues/1964
https://github.com/STEllAR-GROUP/hpx/issues/1962
https://github.com/STEllAR-GROUP/hpx/pull/1963
https://github.com/STEllAR-GROUP/hpx/pull/1961
https://github.com/STEllAR-GROUP/hpx/issues/1960
https://github.com/STEllAR-GROUP/hpx/pull/1959
https://github.com/STEllAR-GROUP/hpx/issues/1958
https://github.com/STEllAR-GROUP/hpx/pull/1957
https://github.com/STEllAR-GROUP/hpx/pull/1956
https://github.com/STEllAR-GROUP/hpx/pull/1955
https://github.com/STEllAR-GROUP/hpx/issues/1954
https://github.com/STEllAR-GROUP/hpx/pull/1953
https://github.com/STEllAR-GROUP/hpx/pull/1952
https://github.com/STEllAR-GROUP/hpx/pull/1951
https://github.com/STEllAR-GROUP/hpx/issues/1950
https://github.com/STEllAR-GROUP/hpx/pull/1949
https://github.com/STEllAR-GROUP/hpx/pull/1948
• PR #1947 - Adding workaround for tagged_tuple
• Issue #1946 - Hang in wait_all() in distributed run
• PR #1945 - Fixed container algorithm tests
• Issue #1944 - assertion `p.destination_locality() == hpx::get_locality()` failed
• PR #1943 - Fix a couple of compile errors with clang
• PR #1942 - Making parcel coalescing functional
• Issue #1941 - Re-enable parcel coalescing
• PR #1940 - Touching up make_future
• PR #1939 - Fixing problems in over-subscription management in the resource manager
• PR #1938 - Cleaning up the use of Boost.Accumulator headers
• PR #1937 - Making sure interval timer is started for aggregating performance counters
• PR #1936 - Tagged results
• PR #1935 - Fix remote async with deferred launch policy
• Issue #1933 - Floating point exception in `statistics_counter<boost::accumulators::tag::mean>::get_counter_value`
• PR #1932 - Removing superfluous includes of `boost/lockfree/detail/branch_hints.hpp`
• PR #1931 - Fix compilation with clang 3.8.0
• Issue #1930 - Missing online documentation for HPX 0.9.11
• PR #1929 - LWG2485: get() should be overloaded for const tuple&
• PR #1928 - Revert “Using ninja for circle-ci builds”
• PR #1927 - Using ninja for circle-ci builds
• PR #1926 - Fixing serialization of std::array
• Issue #1925 - Issues with static HPX libraries

https://github.com/STEllAR-GROUP/hpx/pull/1947
https://github.com/STEllAR-GROUP/hpx/issues/1946
https://github.com/STEllAR-GROUP/hpx/pull/1945
https://github.com/STEllAR-GROUP/hpx/issues/1944
https://github.com/STEllAR-GROUP/hpx/pull/1943
https://github.com/STEllAR-GROUP/hpx/pull/1942
https://github.com/STEllAR-GROUP/hpx/issues/1941
https://github.com/STEllAR-GROUP/hpx/pull/1940
https://github.com/STEllAR-GROUP/hpx/pull/1939
https://github.com/STEllAR-GROUP/hpx/pull/1938
https://github.com/STEllAR-GROUP/hpx/pull/1937
https://github.com/STEllAR-GROUP/hpx/pull/1936
https://github.com/STEllAR-GROUP/hpx/pull/1935
https://github.com/STEllAR-GROUP/hpx/pull/1934
https://github.com/STEllAR-GROUP/hpx/issues/1933
https://github.com/STEllAR-GROUP/hpx/pull/1932
https://github.com/STEllAR-GROUP/hpx/pull/1931
https://github.com/STEllAR-GROUP/hpx/issues/1930
https://github.com/STEllAR-GROUP/hpx/pull/1929
https://github.com/STEllAR-GROUP/hpx/pull/1928
https://github.com/STEllAR-GROUP/hpx/pull/1927
https://github.com/STEllAR-GROUP/hpx/pull/1926
https://github.com/STEllAR-GROUP/hpx/issues/1925
• Issue #1924 - Performance degrading over time
• Issue #1923 - serialization of std::array appears broken in latest commit
• PR #1923 - Container algorithms
• PR #1921 - Tons of smaller quality improvements
• Issue #1920 - Seg fault in hp::serialization::output_archive::add_gid when running octotiger
• Issue #1919 - Intel 15 compiler bug preventing HPX build
• PR #1918 - Address sanitizer fixes
• PR #1917 - Fixing compilation problems of parallel::sort with Intel compilers
• PR #1916 - Making sure code compiles if HPX_WITH_HWLOC=Off
• Issue #1915 - max_cores undefined if HPX_WITH_HWLOC=Off
• PR #1913 - Add utility member functions for partitioned_vector
• PR #1912 - Adding support for invoking actions to dataflow
• PR #1911 - Adding first batch of container algorithms
• PR #1910 - Keep cmake_module_path
• PR #1909 - Fix mpirun with pbs
• PR #1908 - Changing parallel::sort to return the last iterator as proposed by N4560
• PR #1907 - Adding a minimum version for Open MPI
• PR #1906 - Updates to the Release Procedure
• PR #1905 - Fixing #1903
• PR #1904 - Making sure std containers are cleared before serialization loads data
• Issue #1903 - When running octotiger, I get: assertion '(*new_gids_)[gid].size() == 1' failed: HPX(assertion_failure)
• Issue #1902 - Immediate crash when running hp/octotiger with _GLIBCXX_DEBUG defined.
• PR #1901 - Making non-serializable classes non-serializable
• Issue #1900 - Two possible issues with std::list serialization
• PR #1890 - Fixing a problem with credit splitting as revealed by #1898
• Issue #1898 - Accessing component from locality where it was not created segfaults
• PR #1897 - Changing parallel::sort to return the last iterator as proposed by N4560
• Issue #1896 - version 1.0?
• Issue #1895 - Warning comment on numa_allocator is not very clear
• PR #1894 - Add support for compilers that have thread_local
• PR #1893 - Fixing 1890
• PR #1892 - Adds typed future_type for executor_traits
• PR #1891 - Fix wording in certain parallel algorithm docs
• Issue #1890 - Invoking papi counters give segfault
• PR #1889 - Fixing problems as reported by clang-check
• PR #1888 - WIP parallel is_heap
• PR #1887 - Fixed resetting performance counters related to idle-rate, etc
• Issue #1886 - Run hpx with qsub does not work
• PR #1885 - Warning cleaning pass
• PR #1884 - Add missing parallel algorithm header
• PR #1883 - Add feature test for thread_local on Clang for TLS
• PR #1882 - Fix some redundant qualifiers
• Issue #1881 - Unable to compile Octotiger using HPX and Intel MPI on SuperMIC
• Issue #1880 - clang with libc++ on Linux needs TLS case
• PR #1879 - Doc fixes for #1868
• PR #1878 - Simplify functions

https://github.com/STEllAR-GROUP/hpx/issues/1900
https://github.com/STEllAR-GROUP/hpx/pull/1899
https://github.com/STEllAR-GROUP/hpx/issues/1898
https://github.com/STEllAR-GROUP/hpx/pull/1897
https://github.com/STEllAR-GROUP/hpx/pull/1889
https://github.com/STEllAR-GROUP/hpx/pull/1890
https://github.com/STEllAR-GROUP/hpx/pull/1887
https://github.com/STEllAR-GROUP/hpx/pull/1886
https://github.com/STEllAR-GROUP/hpx/pull/1885
https://github.com/STEllAR-GROUP/hpx/pull/1884
https://github.com/STEllAR-GROUP/hpx/pull/1883
https://github.com/STEllAR-GROUP/hpx/pull/1882
https://github.com/STEllAR-GROUP/hpx/pull/1881
https://github.com/STEllAR-GROUP/hpx/pull/1880
https://github.com/STEllAR-GROUP/hpx/pull/1879
https://github.com/STEllAR-GROUP/hpx/pull/1878
• PR #1877 - Removing most usage of Boost.Config
• PR #1876 - Add missing parallel algorithms to algorithm.hpp
• PR #1875 - Simplify callables
• PR #1874 - Address long standing FIXME on using std::unique_ptr with incomplete types
• PR #1873 - Fixing 1871
• PR #1872 - Making sure PBS environment uses specified node list even if no PBS_NODEFILE env is available
• Issue #1871 - Fortran checks should be optional
• PR #1870 - Touch local::mutex
• PR #1869 - Documentation refactoring based off #1868
• PR #1867 - Embrace static_assert
• PR #1866 - Fix #1803 with documentation refactoring
• PR #1865 - Setting OUTPUT_NAME as target properties
• PR #1863 - Use SYSTEM for boost includes
• PR #1862 - Minor cleanups
• PR #1861 - Minor Corrections for Release
• PR #1860 - Fixing hpx gdb script
• Issue #1859 - reset_active_counters resets times and thread counts before some of the counters are evaluated
• PR #1858 - Release V0.9.11
• PR #1857 - removing diskperf example from 9.11 release
• PR #1856 - fix return in packaged_task_base::reset()
• Issue #1842 - Install error: file INSTALL cannot find libhpx_parcel_coalescing.so.0.9.11
• PR #1839 - Adding fedora docs
• PR #1824 - Changing version on master to V0.9.12

3433 https://github.com/STEllAR-GROUP/hpx/pull/1877
3434 https://github.com/STEllAR-GROUP/hpx/pull/1876
3435 https://github.com/STEllAR-GROUP/hpx/pull/1875
3436 https://github.com/STEllAR-GROUP/hpx/pull/1874
3437 https://github.com/STEllAR-GROUP/hpx/pull/1873
3438 https://github.com/STEllAR-GROUP/hpx/pull/1872
3439 https://github.com/STEllAR-GROUP/hpx/issues/1871
3440 https://github.com/STEllAR-GROUP/hpx/pull/1870
3441 https://github.com/STEllAR-GROUP/hpx/pull/1869
3442 https://github.com/STEllAR-GROUP/hpx/pull/1867
3443 https://github.com/STEllAR-GROUP/hpx/pull/1866
3444 https://github.com/STEllAR-GROUP/hpx/pull/1865
3445 https://github.com/STEllAR-GROUP/hpx/pull/1863
3446 https://github.com/STEllAR-GROUP/hpx/pull/1862
3447 https://github.com/STEllAR-GROUP/hpx/pull/1861
3448 https://github.com/STEllAR-GROUP/hpx/pull/1860
3449 https://github.com/STEllAR-GROUP/hpx/issues/1859
3450 https://github.com/STEllAR-GROUP/hpx/pull/1858
3451 https://github.com/STEllAR-GROUP/hpx/pull/1857
3452 https://github.com/STEllAR-GROUP/hpx/pull/1856
3453 https://github.com/STEllAR-GROUP/hpx/issues/1842
3454 https://github.com/STEllAR-GROUP/hpx/pull/1839
3455 https://github.com/STEllAR-GROUP/hpx/pull/1824

2.10. Releases
• PR #1818⁴⁵⁶ - Fixing #1748
• Issue #1815⁴⁵⁷ - seg fault in AGAS
• Issue #1803⁴⁵⁸ - wait_all documentation
• Issue #1796⁴⁵⁹ - Outdated documentation to be revised
• Issue #1759⁴⁶⁰ - glibc munmap_chunk or free(): invalid pointer on SuperMIC
• Issue #1753⁴⁶¹ - HPX performance degrades with time since execution begins
• Issue #1748⁴⁶² - All public HPX headers need to be self contained
• PR #1710⁴⁶³ - How to build HPX with Visual Studio
• Issue #1684⁴⁶⁴ - Race condition when using –hpx:connect?
• PR #1658⁴⁶⁵ - Add serialization for std::set (as there is for std::vector and std::map)
• PR #1641⁴⁶⁶ - Generic client
• Issue #1632⁴⁶⁷ - heartbeat example fails on separate nodes
• PR #1603⁴⁶⁸ - Adds preferred namespace check to inspect tool
• Issue #1559⁴⁶⁹ - Extend inspect tool
• Issue #1523⁴⁷⁰ - Remote async with deferred launch policy never executes
• Issue #1472⁴⁷¹ - Serialization issues
• Issue #1457⁴⁷² - Implement N4392: C++ Latches and Barriers
• PR #1444⁴⁷³ - Enabling usage of moveonly types for component construction
• Issue #1407⁴⁷⁴ - The Intel 13 compiler has failing unit tests
• Issue #1405⁴⁷⁵ - Allow component constructors to take movable only types
• Issue #1265⁴⁷⁶ - Enable dataflow() to be usable with actions
• Issue #1236⁴⁷⁷ - NUMA aware allocators
• Issue #802⁴⁷⁸ - Fix Broken Examples

⁴⁵⁶ https://github.com/STEllAR-GROUP/hpx/pull/1818
⁴⁵⁷ https://github.com/STEllAR-GROUP/hpx/issues/1815
⁴⁵⁸ https://github.com/STEllAR-GROUP/hpx/issues/1803
⁴⁵⁹ https://github.com/STEllAR-GROUP/hpx/issues/1796
⁴⁶⁰ https://github.com/STEllAR-GROUP/hpx/issues/1759
⁴⁶¹ https://github.com/STEllAR-GROUP/hpx/issues/1753
⁴⁶² https://github.com/STEllAR-GROUP/hpx/issues/1748
⁴⁶³ https://github.com/STEllAR-GROUP/hpx/pull/1719
⁴⁶⁴ https://github.com/STEllAR-GROUP/hpx/issues/1684
⁴⁶⁵ https://github.com/STEllAR-GROUP/hpx/pull/1658
⁴⁶⁶ https://github.com/STEllAR-GROUP/hpx/pull/1641
⁴⁶⁷ https://github.com/STEllAR-GROUP/hpx/issues/1632
⁴⁶⁸ https://github.com/STEllAR-GROUP/hpx/pull/1603
⁴⁶⁹ https://github.com/STEllAR-GROUP/hpx/issues/1559
⁴⁷⁰ https://github.com/STEllAR-GROUP/hpx/issues/1523
⁴⁷¹ https://github.com/STEllAR-GROUP/hpx/issues/1472
⁴⁷² https://github.com/STEllAR-GROUP/hpx/issues/1457
⁴⁷³ https://github.com/STEllAR-GROUP/hpx/pull/1444
⁴⁷⁴ https://github.com/STEllAR-GROUP/hpx/pull/1407
⁴⁷⁵ https://github.com/STEllAR-GROUP/hpx/issues/1405
⁴⁷⁶ https://github.com/STEllAR-GROUP/hpx/issues/1265
⁴⁷⁷ https://github.com/STEllAR-GROUP/hpx/issues/1236
⁴⁷⁸ https://github.com/STEllAR-GROUP/hpx/issues/802

Chapter 2. What’s so special about HPX?
• Issue #559\textsuperscript{3479} - Add hpx::migrate facility
• Issue #449\textsuperscript{3480} - Make actions with template arguments usable and add documentation
• Issue #279\textsuperscript{3481} - Refactor addressing_service into a base class and two derived classes
• Issue #224\textsuperscript{3482} - Changing thread state metadata is not thread safe
• Issue #55\textsuperscript{3483} - Uniform syntax for enums should be implemented

2.10.14 \textit{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 choice 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 (\texttt{master}) are prohibited. Any change has to go through a peer review before it will be merged to \texttt{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.

General changes

• We are moving into the direction of unifying managed and simple components. As such, the classes \texttt{hpx::components::component} and \texttt{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\textsuperscript{3484} 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 \texttt{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 \texttt{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:
  
  – \texttt{hpx::components::default_distribution_policy} which tries to place multiple component instances as evenly as possible.

\textsuperscript{3479} https://github.com/STEllAR-GROUP/hpx/issues/559
\textsuperscript{3480} https://github.com/STEllAR-GROUP/hpx/issues/449
\textsuperscript{3481} https://github.com/STEllAR-GROUP/hpx/issues/279
\textsuperscript{3482} https://github.com/STEllAR-GROUP/hpx/issues/224
\textsuperscript{3483} https://github.com/STEllAR-GROUP/hpx/issues/55
\textsuperscript{3484} https://circleci.com/gh/STEllAR-GROUP/hpx
- `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 distribution 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.  

• 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).

• 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`.

• 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 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.

- 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[^488] 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 variables used to configure HPX).

- The get_gid() functions exposed by the component base classes hpx::components::server::simple_component_base, hpx::components::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** - Completely removing external/endian
- **PR #1854** - Don’t pollute CMAKE_CXX_FLAGS through find_package()
- **PR #1853** - Updating CMake configuration to get correct version of TAU library
- **PR #1852** - Fixing Performance Problems with MPI Parcelport
- **PR #1851** - Fixing hpx_add_link_flag() and hpx_remove_link_flag()
- **PR #1850** - Fixing 1836, adding parallel::sort
- **PR #1849** - Fixing configuration for use of more than 64 cores
- **PR #1848** - Change default APEX version for release
- **PR #1847** - Fix client_base::then on release
- **PR #1846** - Removing broken lcos::local::channel from release
- **PR #1845** - Adding example demonstrating a possible safe-object implementation to release
- **PR #1844** - Removing stubs from accumulator examples
- **PR #1843** - Don’t pollute CMAKE_CXX_FLAGS through find_package()
- **PR #1842** - Fixing client_base<>::then
- **PR #1841** - Adding example demonstrating a possible safe-object implementation
- **PR #1838** - Update version rc1
- **PR #1837** - Removing broken lcos::local::channel
- **PR #1835** - Adding explicit move constructor and assignment operator to hpx::lcos::promise
- **PR #1834** - Making hpx::lcos::promise move-only
- **PR #1833** - Adding fedora docs
- **Issue #1832** - hpx::lcos::promise<> must be move-only

References:
- [PR #1855](https://github.com/STEllAR-GROUP/hpx/pull/1855)
- [PR #1854](https://github.com/STEllAR-GROUP/hpx/pull/1854)
- [PR #1853](https://github.com/STEllAR-GROUP/hpx/pull/1853)
- [PR #1852](https://github.com/STEllAR-GROUP/hpx/pull/1852)
- [PR #1851](https://github.com/STEllAR-GROUP/hpx/pull/1851)
- [PR #1850](https://github.com/STEllAR-GROUP/hpx/pull/1850)
- [PR #1849](https://github.com/STEllAR-GROUP/hpx/pull/1849)
- [PR #1848](https://github.com/STEllAR-GROUP/hpx/pull/1848)
- [PR #1847](https://github.com/STEllAR-GROUP/hpx/pull/1847)
- [PR #1846](https://github.com/STEllAR-GROUP/hpx/pull/1846)
- [PR #1845](https://github.com/STEllAR-GROUP/hpx/pull/1845)
- [PR #1844](https://github.com/STEllAR-GROUP/hpx/pull/1844)
- [PR #1843](https://github.com/STEllAR-GROUP/hpx/pull/1843)
- [PR #1842](https://github.com/STEllAR-GROUP/hpx/pull/1842)
- [PR #1841](https://github.com/STEllAR-GROUP/hpx/pull/1841)
- [PR #1838](https://github.com/STEllAR-GROUP/hpx/pull/1838)
- [PR #1837](https://github.com/STEllAR-GROUP/hpx/pull/1837)
- [PR #1835](https://github.com/STEllAR-GROUP/hpx/pull/1835)
- [PR #1834](https://github.com/STEllAR-GROUP/hpx/pull/1834)
- [PR #1833](https://github.com/STEllAR-GROUP/hpx/pull/1833)
- [PR #1832](https://github.com/STEllAR-GROUP/hpx/issues/1832)
- PR #1831 - Fixing resource manager gcc5.2
- PR #1830 - Fix intel13
- PR #1829 - Unbreaking thread test
- PR #1828 - Fixing #1620
- PR #1827 - Fixing a memory management issue for the Parquet application
- Issue #1826 - Memory management issue in hpx::lcos::promise
- PR #1825 - Adding hpx::components::component and hpx::components::component_base
- PR #1823 - Adding git commit id to circleci build
- PR #1822 - applying fixes suggested by clang 3.7
- PR #1821 - Hyperlink fixes
- PR #1820 - added parallel multi-locality sanity test
- PR #1819 - Fixing #1667
- Issue #1817 - Hyperlinks generated by inspect tool are wrong
- PR #1816 - Support hpxrx
- PR #1814 - Fix async to dispatch to the correct locality in all cases
- Issue #1813 - async(launch::... , action(), ...) always invokes locally
- PR #1812 - fixed syntax error in CMakeLists.txt
- PR #1811 - Agas optimizations
- PR #1810 - drop superfluous typedefs
- PR #1809 - Allow HPX to be used as an optional package in 3rd party code
- PR #1808 - Fixing #1723
- PR #1807 - Making sure resolve_localities does not hang during normal operation
- Issue #1806 - Spinlock no longer movable and deletes operator ‘=’, breaks MiniGhost

3510 https://github.com/STEllAR-GROUP/hpx/pull/1831
3511 https://github.com/STEllAR-GROUP/hpx/pull/1830
3512 https://github.com/STEllAR-GROUP/hpx/pull/1829
3513 https://github.com/STEllAR-GROUP/hpx/pull/1828
3514 https://github.com/STEllAR-GROUP/hpx/pull/1827
3515 https://github.com/STEllAR-GROUP/hpx/issues/1826
3516 https://github.com/STEllAR-GROUP/hpx/pull/1825
3517 https://github.com/STEllAR-GROUP/hpx/pull/1823
3518 https://github.com/STEllAR-GROUP/hpx/pull/1822
3519 https://github.com/STEllAR-GROUP/hpx/pull/1821
3520 https://github.com/STEllAR-GROUP/hpx/pull/1820
3521 https://github.com/STEllAR-GROUP/hpx/pull/1819
3522 https://github.com/STEllAR-GROUP/hpx/issues/1817
3523 https://github.com/STEllAR-GROUP/hpx/pull/1816
3524 https://github.com/STEllAR-GROUP/hpx/pull/1814
3525 https://github.com/STEllAR-GROUP/hpx/issues/1813
3526 https://github.com/STEllAR-GROUP/hpx/pull/1812
3527 https://github.com/STEllAR-GROUP/hpx/pull/1811
3528 https://github.com/STEllAR-GROUP/hpx/pull/1810
3529 https://github.com/STEllAR-GROUP/hpx/pull/1809
3530 https://github.com/STEllAR-GROUP/hpx/pull/1808
3531 https://github.com/STEllAR-GROUP/hpx/pull/1807
3532 https://github.com/STEllAR-GROUP/hpx/issues/1806

2.10. Releases
• Issue #1804 - register_with_basename causes hangs
• PR #1801 - Enhanced the inspect tool to take user directly to the problem with hyperlinks
• Issue #1800 - Problems compiling application on smic
• PR #1799 - Fixing cv exceptions
• PR #1798 - Documentation refactoring & updating
• PR #1797 - Updating the activeharmony CMake module
• PR #1795 - Fixing cv
• PR #1794 - Fix connect with hpx::runtime_mode_connect
• PR #1793 - fix a wrong use of HPX_MAX_CPU_COUNT instead of HPX_HAVE_MAX_CPU_COUNT
• PR #1792 - Allow for default constructed parcel instances to be moved
• PR #1791 - Fix connect with hpx::runtime_mode_connect
• Issue #1790 - assertion action_.get() failed: HPX(assertion_failure) when running Octotiger with pull request 1786
• PR #1789 - Fixing discover_counter_types API function
• Issue #1788 - connect with hpx::runtime_mode_connect
• Issue #1787 - discover_counter_types not working
• PR #1786 - Changing addressing_service to use std::unordered_map instead of std::map
• PR #1785 - Fix is_iterator for container algorithms
• PR #1784 - Adding new command line options:
• PR #1783 - Minor changes for APEX support
• PR #1782 - Drop legacy forwarding action traits
• PR #1781 - Attempt to resolve the race between cv::wait_xxx and cv::notify_all
• PR #1780 - Removing serialize_sequence
• PR #1779 - Fixed #1501: hwloc configuration options are wrong for MIC

3533 https://github.com/STEllAR-GROUP/hpx/issues/1804
3534 https://github.com/STEllAR-GROUP/hpx/pull/1801
3535 https://github.com/STEllAR-GROUP/hpx/issues/1800
3536 https://github.com/STEllAR-GROUP/hpx/pull/1799
3537 https://github.com/STEllAR-GROUP/hpx/pull/1798
3538 https://github.com/STEllAR-GROUP/hpx/pull/1797
3539 https://github.com/STEllAR-GROUP/hpx/pull/1795
3540 https://github.com/STEllAR-GROUP/hpx/pull/1794
3541 https://github.com/STEllAR-GROUP/hpx/pull/1793
3542 https://github.com/STEllAR-GROUP/hpx/pull/1792
3543 https://github.com/STEllAR-GROUP/hpx/pull/1791
3544 https://github.com/STEllAR-GROUP/hpx/issues/1790
3545 https://github.com/STEllAR-GROUP/hpx/pull/1789
3546 https://github.com/STEllAR-GROUP/hpx/issues/1788
3547 https://github.com/STEllAR-GROUP/hpx/issues/1787
3548 https://github.com/STEllAR-GROUP/hpx/pull/1786
3549 https://github.com/STEllAR-GROUP/hpx/pull/1785
3550 https://github.com/STEllAR-GROUP/hpx/pull/1784
3551 https://github.com/STEllAR-GROUP/hpx/pull/1783
3552 https://github.com/STEllAR-GROUP/hpx/pull/1782
3553 https://github.com/STEllAR-GROUP/hpx/pull/1781
3554 https://github.com/STEllAR-GROUP/hpx/pull/1780
3555 https://github.com/STEllAR-GROUP/hpx/pull/1779
- PR #1778 - Removing ability to enable/disable parcel handling
- PR #1777 - Completely removing stackless threads
- PR #1776 - Cleaning up util/plugin
- PR #1775 - Agas fixes
- PR #1774 - Action invocation count
- PR #1773 - replaced MSVC variable with WIN32
- PR #1772 - Fixing Problems in MPI parcelport and future serialization.
- PR #1771 - Fixing intel 13 compiler errors related to variadic template template parameters for lcos::when_tests
- PR #1770 - Forwarding decay to std::
- PR #1769 - Add more characters with special regex meaning to the existing patch
- PR #1768 - Adding test for receive_buffer
- PR #1767 - Making sure that uptime counter throws exception on any attempt to be reset
- PR #1766 - Cleaning up code related to throttling scheduler
- PR #1765 - Restricting thread_data to creating only with intrusive_pointers
- PR #1764 - Fixing 1763
- Issue #1763 - UB in thread_data::operator delete
- PR #1762 - Making sure all serialization registries/factories are unique
- PR #1761 - Fixed #1751: hpx::future::wait_for fails a simple test
- PR #1758 - Fixing #1757
- Issue #1757 - pinning not correct using –hpx:bind
- Issue #1756 - compilation error with MinGW
- PR #1755 - Making output serialization const-correct
- Issue #1753 - HPX performance degrades with time since execution begins

3556 https://github.com/STEllAR-GROUP/hpx/pull/1778
3557 https://github.com/STEllAR-GROUP/hpx/pull/1777
3558 https://github.com/STEllAR-GROUP/hpx/pull/1776
3559 https://github.com/STEllAR-GROUP/hpx/pull/1775
3560 https://github.com/STEllAR-GROUP/hpx/pull/1774
3561 https://github.com/STEllAR-GROUP/hpx/pull/1773
3562 https://github.com/STEllAR-GROUP/hpx/pull/1772
3563 https://github.com/STEllAR-GROUP/hpx/pull/1771
3564 https://github.com/STEllAR-GROUP/hpx/pull/1770
3565 https://github.com/STEllAR-GROUP/hpx/pull/1769
3566 https://github.com/STEllAR-GROUP/hpx/pull/1768
3567 https://github.com/STEllAR-GROUP/hpx/pull/1767
3568 https://github.com/STEllAR-GROUP/hpx/pull/1766
3569 https://github.com/STEllAR-GROUP/hpx/pull/1765
3570 https://github.com/STEllAR-GROUP/hpx/pull/1764
3571 https://github.com/STEllAR-GROUP/hpx/issues/1763
3572 https://github.com/STEllAR-GROUP/hpx/pull/1762
3573 https://github.com/STEllAR-GROUP/hpx/pull/1761
3574 https://github.com/STEllAR-GROUP/hpx/pull/1758
3575 https://github.com/STEllAR-GROUP/hpx/issues/1757
3576 https://github.com/STEllAR-GROUP/hpx/issues/1756
3577 https://github.com/STEllAR-GROUP/hpx/pull/1755
3578 https://github.com/STEllAR-GROUP/hpx/issues/1753

2.10. Releases
• Issue #1752\textsuperscript{3579} - Error in AGAS
• Issue #1751\textsuperscript{3580} - hpx::future::wait_for fails a simple test
• PR #1750\textsuperscript{3581} - Removing hpx_fwd.hpp includes
• PR #1749\textsuperscript{3582} - Simplify result_of and friends
• PR #1747\textsuperscript{3583} - Removed superfluous code from message_buffer.hpp
• PR #1746\textsuperscript{3584} - Tuple dependencies
• Issue #1745\textsuperscript{3585} - Broken when_some which takes iterators
• PR #1744\textsuperscript{3586} - Refining archive interface
• PR #1743\textsuperscript{3587} - Fixing when_all when only a single future is passed
• PR #1742\textsuperscript{3588} - Config includes
• PR #1741\textsuperscript{3589} - Os executors
• Issue #1740\textsuperscript{3590} - hpx::promise has some problems
• PR #1739\textsuperscript{3591} - Parallel composition with generic containers
• Issue #1738\textsuperscript{3592} - After building program and successfully linking to a version of hpx DHPX_DIR seems to be ignored
• Issue #1737\textsuperscript{3593} - Uptime problems
• PR #1736\textsuperscript{3594} - added convenience c-tor and begin()/end() to serialize_buffer
• PR #1735\textsuperscript{3595} - Config includes
• PR #1734\textsuperscript{3596} - Fixed #1688: Add timer counters for tfunc_total and exec_total
• Issue #1733\textsuperscript{3597} - Add unit test for hpx/lcos/local/receive_buffer.hpp
• PR #1732\textsuperscript{3598} - Renaming get_os_thread_count
• PR #1731\textsuperscript{3599} - Basename registration
• Issue #1730\textsuperscript{3600} - Use after move of thread_init_data
• PR #1729\textsuperscript{3601} - Rewriting channel based on new gate component

\textsuperscript{3579} https://github.com/STEllAR-GROUP/hpx/issues/1752
\textsuperscript{3580} https://github.com/STEllAR-GROUP/hpx/issues/1751
\textsuperscript{3581} https://github.com/STEllAR-GROUP/hpx/pull/1750
\textsuperscript{3582} https://github.com/STEllAR-GROUP/hpx/pull/1749
\textsuperscript{3583} https://github.com/STEllAR-GROUP/hpx/pull/1747
\textsuperscript{3584} https://github.com/STEllAR-GROUP/hpx/pull/1746
\textsuperscript{3585} https://github.com/STEllAR-GROUP/hpx/issues/1745
\textsuperscript{3586} https://github.com/STEllAR-GROUP/hpx/pull/1744
\textsuperscript{3587} https://github.com/STEllAR-GROUP/hpx/pull/1743
\textsuperscript{3588} https://github.com/STEllAR-GROUP/hpx/pull/1742
\textsuperscript{3589} https://github.com/STEllAR-GROUP/hpx/pull/1741
\textsuperscript{3590} https://github.com/STEllAR-GROUP/hpx/issues/1740
\textsuperscript{3591} https://github.com/STEllAR-GROUP/hpx/pull/1739
\textsuperscript{3592} https://github.com/STEllAR-GROUP/hpx/issues/1738
\textsuperscript{3593} https://github.com/STEllAR-GROUP/hpx/issues/1737
\textsuperscript{3594} https://github.com/STEllAR-GROUP/hpx/pull/1736
\textsuperscript{3595} https://github.com/STEllAR-GROUP/hpx/pull/1735
\textsuperscript{3596} https://github.com/STEllAR-GROUP/hpx/pull/1734
\textsuperscript{3597} https://github.com/STEllAR-GROUP/hpx/issues/1733
\textsuperscript{3598} https://github.com/STEllAR-GROUP/hpx/pull/1732
\textsuperscript{3599} https://github.com/STEllAR-GROUP/hpx/pull/1731
\textsuperscript{3600} https://github.com/STEllAR-GROUP/hpx/issues/1730
\textsuperscript{3601} https://github.com/STEllAR-GROUP/hpx/pull/1729

Chapter 2. What’s so special about \textit{HPX}?
• PR #1728 - Fixing #1722
• PR #1723 - Fixing compile problems with apply_colocated
• PR #1726 - Apex integration
• PR #1725 - fixed test timeouts
• PR #1724 - Renaming vector
• Issue #1723 - Drop support for intel compilers and gcc 4.4 based standard libs
• Issue #1722 - Add support for detecting non-ready futures before serialization
• PR #1721 - Unifying parallel executors, initializing from launch policy
• PR #1720 - dropped superfluous typedef
• Issue #1718 - Windows 10 x64, VS 2015 - Unknown CMake command “add_hpx_pseudo_target”.
• PR #1717 - Timed executor traits for thread-executors
• PR #1716 - serialization of arrays didn’t work with non-pod types. fixed
• PR #1715 - List serialization
• PR #1714 - changing misspellings
• PR #1713 - Fixed distribution policy executors
• PR #1712 - Moving library detection to be executed after feature tests
• PR #1711 - Simplify parcel
• PR #1710 - Compile only tests
• PR #1709 - Implemented timed executors
• PR #1708 - Implement parallel::executor_traits for thread-executors
• PR #1707 - Various fixes to threads::executors to make custom schedulers work
• PR #1706 - Command line option --hpx:cores does not work as expected
• Issue #1705 - command line option --hpx:cores does not work as expected
• PR #1704 - vector deserialization is speeded up a little
• PR #1703 - Fixing shared_mutes
• Issue #1702 - Shared_mutex does not compile with no_mutex cond_var
• PR #1701 - Add distribution_policy_executor
• PR #1700 - Executor parameters
• PR #1699 - Readers writer lock
• PR #1698 - Remove leftovers
• PR #1697 - Fixing held locks
• PR #1696 - Modified Scan Partitioner for Algorithms
• PR #1695 - This thread executors
• PR #1694 - Fixed #1688: Add timer counters for tfunc_total and exec_total
• PR #1693 - Fix #1691: is_executor template specification fails for inherited executors
• PR #1692 - Fixed #1662: Possible exception source in coalescing_message_handler
• Issue #1691 - is_executor template specification fails for inherited executors
• PR #1690 - added macro for non-intrusive serialization of classes without a default c-tor
• PR #1689 - Replace value_or_error with custom storage, unify future_data state
• Issue #1688 - Add timer counters for tfunc_total and exec_total
• PR #1687 - Fixed interval timer
• PR #1686 - Fixing cmake warnings about not existing pseudo target dependencies
• PR #1685 - Converting partitioners to use bulk async execute
• PR #1683 - Adds a tool for inspect that checks for character limits
• PR #1682 - Change project name to (uppercase) HPX
• PR #1681 - Counter shortnames

3625 https://github.com/STEllAR-GROUP/hpx/pull/1704
3626 https://github.com/STEllAR-GROUP/hpx/pull/1703
3627 https://github.com/STEllAR-GROUP/hpx/issues/1702
3628 https://github.com/STEllAR-GROUP/hpx/pull/1701
3629 https://github.com/STEllAR-GROUP/hpx/pull/1700
3630 https://github.com/STEllAR-GROUP/hpx/pull/1699
3631 https://github.com/STEllAR-GROUP/hpx/pull/1698
3632 https://github.com/STEllAR-GROUP/hpx/pull/1697
3633 https://github.com/STEllAR-GROUP/hpx/pull/1696
3634 https://github.com/STEllAR-GROUP/hpx/pull/1695
3635 https://github.com/STEllAR-GROUP/hpx/pull/1694
3636 https://github.com/STEllAR-GROUP/hpx/pull/1693
3637 https://github.com/STEllAR-GROUP/hpx/pull/1692
3638 https://github.com/STEllAR-GROUP/hpx/issues/1691
3639 https://github.com/STEllAR-GROUP/hpx/pull/1690
3640 https://github.com/STEllAR-GROUP/hpx/pull/1689
3641 https://github.com/STEllAR-GROUP/hpx/issues/1688
3642 https://github.com/STEllAR-GROUP/hpx/pull/1687
3643 https://github.com/STEllAR-GROUP/hpx/pull/1686
3644 https://github.com/STEllAR-GROUP/hpx/pull/1685
3645 https://github.com/STEllAR-GROUP/hpx/pull/1683
3646 https://github.com/STEllAR-GROUP/hpx/pull/1682
3647 https://github.com/STEllAR-GROUP/hpx/pull/1681

1760 Chapter 2. What’s so special about HPX?
• PR #1680 - Extended Non-intrusive Serialization to Ease Usage for Library Developers
• PR #1679 - Working on 1544: More executor changes
• PR #1678 - Transpose fixes
• PR #1677 - Improve Boost compatibility check
• PR #1676 - 1d stencil fix
• Issue #1675 - hpx project name is not HPX
• PR #1674 - Fixing the MPI parcelport
• PR #1673 - added move semantics to map/vector deserialization
• PR #1672 - Vs2015 await
• PR #1671 - Adapt transform for #1668
• PR #1670 - Started to work on #1668
• PR #1669 - Add this_thread_executors
• Issue #1667 - Apple build instructions in docs are out of date
• PR #1666 - Apex integration
• PR #1665 - Fixes an error with the whitespace check that showed the incorrect location of the error
• Issue #1664 - Inspect tool found incorrect endlne whitespace
• PR #1663 - Improve use of locks
• Issue #1662 - Possible exception source in coalescing_message_handler
• PR #1661 - Added support for 128bit number serialization
• PR #1660 - Serialization 128bits
• PR #1659 - Implemented inner_product and adjacent_diff algos
• PR #1658 - Add serialization for std::set (as there is for std::vector and std::map)
• PR #1657 - Use of shared_ptr in io_service_pool changed to unique_ptr

https://github.com/STEllAR-GROUP/hpx/pull/1680
https://github.com/STEllAR-GROUP/hpx/pull/1679
https://github.com/STEllAR-GROUP/hpx/pull/1678
https://github.com/STEllAR-GROUP/hpx/pull/1677
https://github.com/STEllAR-GROUP/hpx/pull/1676
https://github.com/STEllAR-GROUP/hpx/pull/1675
https://github.com/STEllAR-GROUP/hpx/pull/1674
https://github.com/STEllAR-GROUP/hpx/pull/1673
https://github.com/STEllAR-GROUP/hpx/pull/1672
https://github.com/STEllAR-GROUP/hpx/pull/1671
https://github.com/STEllAR-GROUP/hpx/pull/1670
https://github.com/STEllAR-GROUP/hpx/pull/1669
https://github.com/STEllAR-GROUP/hpx/issues/1667
https://github.com/STEllAR-GROUP/hpx/pull/1666
https://github.com/STEllAR-GROUP/hpx/pull/1665
https://github.com/STEllAR-GROUP/hpx/pull/1664
https://github.com/STEllAR-GROUP/hpx/pull/1663
https://github.com/STEllAR-GROUP/hpx/pull/1662
https://github.com/STEllAR-GROUP/hpx/pull/1661
https://github.com/STEllAR-GROUP/hpx/pull/1660
https://github.com/STEllAR-GROUP/hpx/pull/1659
https://github.com/STEllAR-GROUP/hpx/pull/1658
https://github.com/STEllAR-GROUP/hpx/pull/1657
• Issue #1656 - 1d_stencil codes all have wrong factor
• PR #1654 - When using runtime_mode_connect, find the correct localhost public ip address
• PR #1652 - Fixing 1617
• PR #1654 - Remove traits::action_may_require_id_splitting
• PR #1651 - Fixed performance counters related to AGAS cache timings
• PR #1650 - Remove leftovers of traits::type_size
• PR #1649 - Shorten target names on Windows to shorten used path names
• PR #1648 - Fixing problems introduced by merging #1623 for older compilers
• PR #1647 - Simplify running automatic builds on Windows
• Issue #1646 - Cache insert and update performance counters are broken
• Issue #1644 - Remove leftovers of traits::type_size
• Issue #1643 - Remove traits::action_may_require_id_splitting
• PR #1642 - Adds spell checker to the inspect tool for qbk and doxygen comments
• PR #1640 - First step towards fixing 688
• PR #1639 - Re-apply remaining changes from limit_dataflow_recursion branch
• PR #1638 - This fixes possible deadlock in the test ignore_while_locked_1485
• PR #1637 - Fixing hpx::wait_all() invoked with two vector<future<T>>
• PR #1636 - Partially re-apply changes from limit_dataflow_recursion branch
• PR #1635 - Adding missing test for #1572
• PR #1634 - Revert “Limit recursion-depth in dataflow to a configurable constant”
• PR #1633 - Add command line option to ignore batch environment
• PR #1631 - hpx::lcos::queue exhibits strange behavior
• PR #1630 - Fixed endline_whitespace_check.cpp to detect lines with only whitespace
• **Issue #1629** - Inspect trailing whitespace checker problem
• **PR #1628** - Removed meaningless const qualifiers. Minor icpc fix.
• **PR #1627** - Fixing the queue LCO and add example demonstrating its use
• **PR #1626** - Deprecating get_gid(), add get_id() and get_unmanaged_id()
• **PR #1625** - Allowing to specify whether to send credits along with message
• **Issue #1624** - Lifetime issue
• **Issue #1623** - hpx::wait_all() invoked with two vector<future<T>> fails
• **PR #1622** - Executor partitioners
• **PR #1621** - Clean up coroutines implementation
• **Issue #1620** - Revert #1535
• **PR #1619** - Fix result type calculation for hpx::make_continuation
• **PR #1618** - Fixing RDTSC on Xeon/Phi
• **Issue #1617** - hpx cmake not working when run as a subproject
• **Issue #1616** - cmake problem resulting in RDTSC not working correctly for Xeon Phi creates very strange results for duration counters
• **Issue #1615** - hpx::make_continuation requires input and output to be the same
• **PR #1614** - Fixed remove copy test
• **Issue #1613** - Dataflow causes stack overflow
• **PR #1612** - Modified foreach partitioner to use bulk execute
• **PR #1611** - Limit recursion-depth in dataflow to a configurable constant
• **PR #1610** - Increase timeout for CircleCI
• **PR #1609** - Refactoring thread manager, mainly extracting thread pool
• **PR #1608** - Fixed running multiple localities without localities parameter
• **PR #1607** - More algorithm fixes to adjacentfind

---

[3694] https://github.com/STEllAR-GROUP/hpx/issues/1629
[3695] https://github.com/STEllAR-GROUP/hpx/pull/1628
[3696] https://github.com/STEllAR-GROUP/hpx/pull/1627
[3697] https://github.com/STEllAR-GROUP/hpx/pull/1626
[3698] https://github.com/STEllAR-GROUP/hpx/pull/1625
[3699] https://github.com/STEllAR-GROUP/hpx/pull/1624
[3700] https://github.com/STEllAR-GROUP/hpx/pull/1623
[3701] https://github.com/STEllAR-GROUP/hpx/pull/1622
[3702] https://github.com/STEllAR-GROUP/hpx/pull/1621
[3703] https://github.com/STEllAR-GROUP/hpx/pull/1620
[3704] https://github.com/STEllAR-GROUP/hpx/pull/1619
[3705] https://github.com/STEllAR-GROUP/hpx/pull/1618
[3706] https://github.com/STEllAR-GROUP/hpx/pull/1617
[3707] https://github.com/STEllAR-GROUP/hpx/pull/1616
[3708] https://github.com/STEllAR-GROUP/hpx/pull/1615
[3709] https://github.com/STEllAR-GROUP/hpx/pull/1614
[3710] https://github.com/STEllAR-GROUP/hpx/pull/1613
[3711] https://github.com/STEllAR-GROUP/hpx/pull/1612
[3712] https://github.com/STEllAR-GROUP/hpx/pull/1611
[3713] https://github.com/STEllAR-GROUP/hpx/pull/1610
[3714] https://github.com/STEllAR-GROUP/hpx/pull/1609
[3715] https://github.com/STEllAR-GROUP/hpx/pull/1608
[3716] https://github.com/STEllAR-GROUP/hpx/pull/1607
• Issue #1606[^1606] - Running without localities parameter binds to bogus port range
• Issue #1605[^1605] - Too many serializations
• PR #1604[^1604] - Changes the HPX image into a hyperlink
• PR #1601[^1601] - Fixing problems with remove_copy algorithm tests
• PR #1600[^1600] - Actions with ids cleanup
• PR #1599[^1599] - Duplicate binding of global ids should fail
• PR #1598[^1598] - Fixing array access
• PR #1597[^1597] - Improved the reliability of connecting/disconnecting localities
• Issue #1596[^1596] - Duplicate id binding should fail
• PR #1595[^1595] - Fixing more cmake config constants
• PR #1594[^1594] - Fixing preprocessor constant used to enable C++11 chrono
• PR #1593[^1593] - Adding operator() for hpx::launch
• Issue #1592[^1592] - Error (typo) in the docs
• Issue #1590[^1590] - CMake fails when CMAKE_BINARY_DIR contains ‘+’.
• Issue #1589[^1589] - Disconnecting a locality results in segfault using heartbeat example
• PR #1588[^1588] - Fix doc string for config option HPX_WITH_EXAMPLES
• PR #1586[^1586] - Fixing 1493
• PR #1585[^1585] - Additional Check for Inspect Tool to detect Endline Whitespace
• Issue #1584[^1584] - Clean up coroutines implementation
• PR #1583[^1583] - Adding a check for end line whitespace
• PR #1582[^1582] - Attempt to fix assert firing after scheduling loop was exited
• PR #1581[^1581] - Fixed adjacentfind_binary test
• PR #1580[^1580] - Prevent some of the internal cmake lists from growing indefinitely

[^1606]: https://github.com/STEllAR-GROUP/hpx/issues/1606
[^1605]: https://github.com/STEllAR-GROUP/hpx/issues/1605
[^1604]: https://github.com/STEllAR-GROUP/hpx/pull/1604
[^1601]: https://github.com/STEllAR-GROUP/hpx/pull/1601
[^1600]: https://github.com/STEllAR-GROUP/hpx/pull/1600
[^1599]: https://github.com/STEllAR-GROUP/hpx/pull/1599
[^1598]: https://github.com/STEllAR-GROUP/hpx/pull/1598
[^1596]: https://github.com/STEllAR-GROUP/hpx/pull/1597
[^1595]: https://github.com/STEllAR-GROUP/hpx/pull/1596
[^1592]: https://github.com/STEllAR-GROUP/hpx/pull/1595
[^1594]: https://github.com/STEllAR-GROUP/hpx/pull/1594
[^1590]: https://github.com/STEllAR-GROUP/hpx/pull/1593
[^1593]: https://github.com/STEllAR-GROUP/hpx/pull/1592
[^1597]: https://github.com/STEllAR-GROUP/hpx/pull/1590
[^1589]: https://github.com/STEllAR-GROUP/hpx/pull/1589
[^1586]: https://github.com/STEllAR-GROUP/hpx/pull/1588
[^1585]: https://github.com/STEllAR-GROUP/hpx/pull/1586
[^1583]: https://github.com/STEllAR-GROUP/hpx/pull/1585
[^1584]: https://github.com/STEllAR-GROUP/hpx/pull/1584
[^1582]: https://github.com/STEllAR-GROUP/hpx/pull/1583
[^1581]: https://github.com/STEllAR-GROUP/hpx/pull/1582
[^1580]: https://github.com/STEllAR-GROUP/hpx/pull/1581
• PR #1579 [^3740] - Removing type_size trait, replacing it with special archive type
• Issue #1578 [^3741] - Remove demangle_helper
• PR #1577 [^3742] - Get ptr problems
• Issue #1576 [^3743] - Refactor async, dataflow, and future::then
• PR #1575 [^3744] - Fixing tests for parallel rotate
• PR #1574 [^3745] - Cleaning up schedulers
• PR #1573 [^3746] - Fixing thread pool executor
• PR #1572 [^3747] - Fixing number of configured localities
• PR #1571 [^3748] - Reimplement decay
• PR #1570 [^3749] - Refactoring async, apply, and dataflow APIs
• PR #1569 [^3750] - Changed range for mach-o library lookup
• PR #1568 [^3751] - Mark decltype support as required
• PR #1567 [^3752] - Removed const from algorithms
• Issue #1566 [^3753] - CMAKE Configuration Test Failures for clang 3.5 on debian
• PR #1565 [^3754] - Dylib support
• PR #1564 [^3755] - Converted partitioners and some algorithms to use executors
• PR #1563 [^3756] - Fix several #includes for Boost.Preprocessor
• PR #1562 [^3757] - Adding configuration option disabling/enabling all message handlers
• PR #1561 [^3758] - Removed all occurrences of boost::move replacing it with std::move
• Issue #1560 [^3759] - Leftover HPX_REGISTER_ACTION_DECLARATION_2
• PR #1558 [^3760] - Revisit async/apply SFINAE conditions
• PR #1557 [^3761] - Removing type_size trait, replacing it with special archive type
• PR #1556 [^3762] - Executor algorithms

[^3740]: https://github.com/STEllAR-GROUP/hpx/pull/1579
[^3741]: https://github.com/STEllAR-GROUP/hpx/issues/1578
[^3742]: https://github.com/STEllAR-GROUP/hpx/pull/1577
[^3743]: https://github.com/STEllAR-GROUP/hpx/issues/1576
[^3744]: https://github.com/STEllAR-GROUP/hpx/pull/1575
[^3745]: https://github.com/STEllAR-GROUP/hpx/pull/1574
[^3746]: https://github.com/STEllAR-GROUP/hpx/pull/1573
[^3747]: https://github.com/STEllAR-GROUP/hpx/pull/1572
[^3748]: https://github.com/STEllAR-GROUP/hpx/pull/1571
[^3749]: https://github.com/STEllAR-GROUP/hpx/pull/1570
[^3750]: https://github.com/STEllAR-GROUP/hpx/pull/1569
[^3751]: https://github.com/STEllAR-GROUP/hpx/pull/1568
[^3752]: https://github.com/STEllAR-GROUP/hpx/pull/1567
[^3753]: https://github.com/STEllAR-GROUP/hpx/issues/1566
[^3754]: https://github.com/STEllAR-GROUP/hpx/pull/1565
[^3755]: https://github.com/STEllAR-GROUP/hpx/pull/1564
[^3756]: https://github.com/STEllAR-GROUP/hpx/pull/1563
[^3757]: https://github.com/STEllAR-GROUP/hpx/pull/1562
[^3758]: https://github.com/STEllAR-GROUP/hpx/pull/1561
[^3759]: https://github.com/STEllAR-GROUP/hpx/pull/1560
[^3760]: https://github.com/STEllAR-GROUP/hpx/pull/1558
[^3761]: https://github.com/STEllAR-GROUP/hpx/pull/1557
[^3762]: https://github.com/STEllAR-GROUP/hpx/pull/1556

2.10. Releases
• PR #1555 - Remove the necessity to specify archive flags on the receiving end
• PR #1554 - Removing obsolete Boost.Serialization macros
• PR #1553 - Properly fix HPX_DEFINE_*_ACTION macros
• PR #1552 - Fixed algorithms relying on copy_if implementation
• PR #1551 - Pxs - Modifying FindOrangeFS.cmake based on OrangeFS 2.9.X
• Issue #1550 - Passing plain identifier inside HPX_DEFINEPLAIN_ACTION_1
• PR #1549 - Fixing intel14/libstdc++4.4
• PR #1548 - Moving raw_ptr to detail namespace
• PR #1547 - Adding support for executors to future.then
• PR #1546 - Executor traits result types
• PR #1545 - Integrate executors with dataflow
• PR #1543 - Fix potential zero-copy for primarynamespace::bulk_service_async et.al.
• PR #1542 - Merging HPX0.9.10 into pxfs branch
• PR #1541 - Removed stale cmake tests, unused since the great cmake refactoring
• PR #1540 - Fix idle-rate on platforms without TSC
• PR #1539 - Reporting situation if zero-copy-serialization was performed by a parcel generated from a plain apply/async
• PR #1538 - Changed return type of bulk executors and added test
• Issue #1537 - Incorrect cpuid config tests
• PR #1536 - Changed return type of bulk executors and added test
• PR #1535 - Make sure promise::get_gid() can be called more than once
• PR #1534 - Fixed async_callback with bound callback
• PR #1533 - Updated the link in the documentation to a publically-accessible URL
• PR #1532 - Make sure sync primitives are not copyable nor movable

https://github.com/STEllAR-GROUP/hpx/pull/1555
https://github.com/STEllAR-GROUP/hpx/pull/1554
https://github.com/STEllAR-GROUP/hpx/pull/1553
https://github.com/STEllAR-GROUP/hpx/pull/1552
https://github.com/STEllAR-GROUP/hpx/pull/1551
https://github.com/STEllAR-GROUP/hpx/pull/1550
https://github.com/STEllAR-GROUP/hpx/pull/1549
https://github.com/STEllAR-GROUP/hpx/pull/1548
https://github.com/STEllAR-GROUP/hpx/pull/1547
https://github.com/STEllAR-GROUP/hpx/pull/1546
https://github.com/STEllAR-GROUP/hpx/pull/1545
https://github.com/STEllAR-GROUP/hpx/pull/1543
https://github.com/STEllAR-GROUP/hpx/pull/1542
https://github.com/STEllAR-GROUP/hpx/pull/1541
https://github.com/STEllAR-GROUP/hpx/pull/1540
https://github.com/STEllAR-GROUP/hpx/pull/1539
https://github.com/STEllAR-GROUP/hpx/pull/1538
https://github.com/STEllAR-GROUP/hpx/pull/1537
https://github.com/STEllAR-GROUP/hpx/pull/1536
https://github.com/STEllAR-GROUP/hpx/pull/1535
https://github.com/STEllAR-GROUP/hpx/pull/1534
https://github.com/STEllAR-GROUP/hpx/pull/1533
https://github.com/STEllAR-GROUP/hpx/pull/1532
• PR #1531 - Fix unwrapped issue with future ranges of void type
• PR #1530 - Serialization complex
• Issue #1528 - Unwrapped issue with future<void>
• Issue #1527 - HPX does not build with Boost 1.58.0
• PR #1526 - Added support for boost.multi_array serialization
• PR #1525 - Properly handle deferred futures, fixes #1506
• PR #1524 - Making sure invalid action argument types generate clear error message
• Issue #1522 - Need serialization support for boost multi array
• Issue #1521 - Remote async and zero-copy serialization optimizations don’t play well together
• PR #1520 - Fixing UB while registering polymorphic classes for serialization
• PR #1519 - Making detail::condition_variable safe to use
• PR #1518 - Fix when_some bug missing indices in its result
• Issue #1517 - Typo may affect CMake build system tests
• PR #1516 - Fixing Posix context
• PR #1515 - Fixing Posix context
• PR #1514 - Correct problems with loading dynamic components
• PR #1513 - Fixing intel glibc4 4
• Issue #1508 - memory and papi counters do not work
• Issue #1507 - Unrecognized Command Line Option Error causing exit status 0
• Issue #1506 - Properly handle deferred futures
• PR #1505 - Adding #include - would not compile without this
• Issue #1502 - boost::filesystem::exists throws unexpected exception
• Issue #1501 - hwloc configuration options are wrong for MIC
• PR #1504 - Making sure boost::filesystem::exists() does not throw
• PR #1500 - Exit application on --hpx:version/-v and --hpx:info
• PR #1498 - Extended task block
• PR #1497 - Unique ptr serialization
• PR #1496 - Unique ptr serialization (closed)
• PR #1495 - Switching circleci build type to debug
• Issue #1494 - --hpx:version/-v does not exit after printing version information
• Issue #1493 - add an hpx_ prefix to libraries and components to avoid name conflicts
• Issue #1492 - Define and ensure limitations for arguments to async/apply
• PR #1489 - Enable idle rate counter on demand
• PR #1488 - Made sure detail::condition_variable can be safely destroyed
• PR #1487 - Introduced default (main) template implementation for ignore_while_checking
• PR #1486 - Add HPX inspect tool
• Issue #1485 - ignore_while_locked doesn’t support all Lockable types
• PR #1484 - Docker image generation
• PR #1483 - Move external endian library into HPX
• PR #1482 - Actions with integer type ids
• Issue #1481 - Sync primitives safe destruction
• Issue #1480 - Move external/boost/endian into hpx/util
• Issue #1478 - Boost inspect violations
• PR #1479 - Adds serialization for arrays; some further/minor fixes
• PR #1477 - Fixing problems with the Intel compiler using a GCC 4.4 std library
• PR #1476 - Adding hpx::lcos::latch and hpx::lcos::local::latch

3809 https://github.com/STEllAR-GROUP/hpx/pull/1504
3810 https://github.com/STEllAR-GROUP/hpx/pull/1500
3811 https://github.com/STEllAR-GROUP/hpx/pull/1498
3812 https://github.com/STEllAR-GROUP/hpx/pull/1497
3813 https://github.com/STEllAR-GROUP/hpx/pull/1496
3814 https://github.com/STEllAR-GROUP/hpx/pull/1495
3815 https://github.com/STEllAR-GROUP/hpx/issues/1494
3816 https://github.com/STEllAR-GROUP/hpx/issues/1493
3817 https://github.com/STEllAR-GROUP/hpx/issues/1492
3818 https://github.com/STEllAR-GROUP/hpx/pull/1489
3819 https://github.com/STEllAR-GROUP/hpx/pull/1488
3820 https://github.com/STEllAR-GROUP/hpx/pull/1487
3821 https://github.com/STEllAR-GROUP/hpx/pull/1486
3822 https://github.com/STEllAR-GROUP/hpx/issues/1485
3823 https://github.com/STEllAR-GROUP/hpx/pull/1484
3824 https://github.com/STEllAR-GROUP/hpx/pull/1483
3825 https://github.com/STEllAR-GROUP/hpx/pull/1482
3826 https://github.com/STEllAR-GROUP/hpx/pull/1481
3827 https://github.com/STEllAR-GROUP/hpx/pull/1480
3828 https://github.com/STEllAR-GROUP/hpx/issues/1478
3829 https://github.com/STEllAR-GROUP/hpx/pull/1479
3830 https://github.com/STEllAR-GROUP/hpx/pull/1477
3831 https://github.com/STEllAR-GROUP/hpx/pull/1476

Chapter 2. What’s so special about HPX?
- Issue #1475 - Boost inspect violations
- PR #1473 - Fixing action move tests
- Issue #1471 - Sync primitives should not be movable
- PR #1470 - Removing `hpx::util::polymorphic_factory`
- PR #1469 - Fixed container creation
- Issue #1467 - HPX application fail during finalization
- Issue #1466 - HPX doesn’t pick up Torque’s nodelfile on SuperMIC
- Issue #1464 - HPX option for pre and post bootstrap performance counters
- PR #1463 - Replacing `async_colocated(id, ...) with async(colocated(id), ...)
- PR #1462 - Consolidated task_region with N4411
- PR #1461 - Consolidate inconsistent CMake option names
- Issue #1460 - Which malloc is actually used? or at least which one is HPX built with
- Issue #1459 - Make cmake configure step fail explicitly if compiler version is not supported
- Issue #1458 - Update `parallel::task_region` with N4411
- PR #1456 - Consolidating `new_<>()`
- Issue #1455 - Replace `async_colocated(id, ...) with async(colocated(id), ...)
- PR #1454 - Removed harmful std::moves from return statements
- PR #1453 - Use range-based for-loop instead of Boost.ForEach
- PR #1452 - C++ feature tests
- PR #1451 - When serializing, pass archive flags to traits::get_type_size
- Issue #1450 - traits::get_type_size needs archive flags to enable zero_copy optimizations
- Issue #1449 - “couldn’t create performance counter” - AGAS
- Issue #1448 - Replace distributing factories with `new_<T[]>({...})`

2.10. Releases

1769
• PR #1447 - Removing obsolete remote_object component
• PR #1446 - Hpx serialization
• PR #1445 - Replacing travis with circleci
• PR #1443 - Always stripping HPX command line arguments before executing start function
• PR #1442 - Adding --hpx:bind=none to disable thread affinities
• Issue #1439 - Libraries get linked in multiple times, RPATH is not properly set
• PR #1438 - Removed superfluous typedefs
• Issue #1437 - hpx::init() should strip HPX-related flags from argv
• Issue #1436 - Add strong scaling option to htxs
• PR #1435 - Adding async_cb, async_continue_cb, and async_colocated_cb
• PR #1434 - Added missing install rule, removed some dead CMake code
• PR #1433 - Add GitExternal and SubProject cmake scripts from eyescale/cmake repo
• Issue #1432 - Add command line flag to disable thread pinning
• PR #1431 - Fix #1423
• Issue #1430 - Inconsistent CMake option names
• Issue #1429 - Configure setting HPX_HAVE_PARCELPORT_MPI is ignored
• PR #1428 - Fixes #1419 (closed)
• PR #1427 - Adding stencil_iterator and transform_iterator
• PR #1426 - Fixes #1419
• PR #1425 - During serialization memory allocation should honour allocator chunk size
• Issue #1424 - chunk allocation during serialization does not use memory pool/allocator chunk size
• Issue #1423 - Remove HPX_STD_UNIQUE_PTR
• Issue #1422 - hpx::threads=all allocates too many os threads
• PR #1420 - added .travis.yml
• Issue #1419 - Unify enums: hpx::runtime::state and hpx::state
• PR #1416 - Adding travis builder
• Issue #1414 - Correct directory for dispatch_gcc46.hpp iteration
• Issue #1410 - Set operation algorithms
• Issue #1389 - Parallel algorithms relying on scan partitioner break for small number of elements
• Issue #1325 - Exceptions thrown during parcel handling are not handled correctly
• Issue #1315 - Errors while running performance tests
• Issue #1309 - hpx::vector partitions are not easily extendable by applications
• PR #1300 - Added serialization/de-serialization to examples.tuplespace
• Issue #1251 - hpx::threads::get_thread_count doesn’t consider pending threads
• Issue #1008 - Decrease in application performance overtime; occasional spikes of major slowdown
• Issue #1001 - Zero copy serialization raises assert
• Issue #721 - Make HPX usable for Xeon Phi
• Issue #524 - Extend scheduler to support threads which can’t be stolen

2.10.15 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

3878 https://github.com/STEllAR-GROUP/hpx/pull/1420
3879 https://github.com/STEllAR-GROUP/hpx/issues/1419
3880 https://github.com/STEllAR-GROUP/hpx/pull/1416
3881 https://github.com/STEllAR-GROUP/hpx/issues/1414
3882 https://github.com/STEllAR-GROUP/hpx/issues/1410
3883 https://github.com/STEllAR-GROUP/hpx/issues/1389
3884 https://github.com/STEllAR-GROUP/hpx/issues/1325
3885 https://github.com/STEllAR-GROUP/hpx/issues/1315
3886 https://github.com/STEllAR-GROUP/hpx/issues/1309
3887 https://github.com/STEllAR-GROUP/hpx/pull/1300
3888 https://github.com/STEllAR-GROUP/hpx/issues/1251
3889 https://github.com/STEllAR-GROUP/hpx/issues/1008
3890 https://github.com/STEllAR-GROUP/hpx/issues/1001
3891 https://github.com/STEllAR-GROUP/hpx/issues/721
3892 https://github.com/STEllAR-GROUP/hpx/issues/524
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).

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 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.

- 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 - Internal shared_future serialization copies
- Issue #1399 - Build takes unusually long time...
- Issue #1398 - Tests using the scan partitioner are broken on at least gcc 4.7 and intel compiler
- Issue #1397 - Completely remove hpx::unique_future
- Issue #1396 - Parallel scan algorithms with different initial values

3893 https://github.com/STEllAR-GROUP/hpx/issues/839
3894 https://github.com/STEllAR-GROUP/hpx/issues/1338
3895 https://github.com/STEllAR-GROUP/hpx/issues/1402
3896 https://github.com/STEllAR-GROUP/hpx/issues/1399
3897 https://github.com/STEllAR-GROUP/hpx/issues/1398
3898 https://github.com/STEllAR-GROUP/hpx/issues/1397
3899 https://github.com/STEllAR-GROUP/hpx/issues/1396
• Issue #1395 - Race Condition - 1d_stencil_8 - SuperMIC
• Issue #1394 - “suspending thread while at least one lock is being held” - 1d_stencil_8 - SuperMIC
• Issue #1393 - SEGFAULT in 1d_stencil_8 on SuperMIC
• Issue #1392 - Fixing #1168
• Issue #1391 - Parallel Algorithms for scan partitioner for small number of elements
• Issue #1387 - Failure with more than 4 localities
• Issue #1386 - Dispatching unhandled exceptions to outer user code
• Issue #1385 - Adding Copy algorithms, fixing parallel::copy_if
• Issue #1384 - Fixing 1325
• Issue #1383 - Fixed #504: Refactor Dataflow LCO to work with futures, this removes the dataflow component as it is obsolete
• Issue #1382 - is_sorted, is_sorted_until and is_partitioned algorithms
• Issue #1381 - fix for CMake versions prior to 3.1
• Issue #1380 - resolved warning in CMake 3.1 and newer
• Issue #1379 - Compilation error with papi
• Issue #1378 - Towards safer migration
• Issue #1377 - HPXConfig.cmake should include TCMALLOC_LIBRARY and TCMALLOC_INCLUDE_DIR
• Issue #1376 - Warning on uninitialized member
• Issue #1375 - Fixing 1163
• Issue #1374 - Fixing the MSVC 12 release builder
• Issue #1373 - Modifying parallel search algorithm for zero length searches
• Issue #1372 - Modifying parallel search algorithm for zero length searches
• Issue #1371 - Avoid holding a lock during agas::inref while doing a credit split
• Issue #1370 - --hpx::bind throws unexpected error
• Issue #1369 - Getting rid of (void) in loops
• Issue #1368 - Variadic templates support for tuple
• Issue #1367 - One last batch of variadic templates support
• Issue #1366 - Fixing symbolic namespace hang
• Issue #1365 - More held locks
• Issue #1364 - Add counters 1363
• Issue #1363 - Add thread overhead counters
• Issue #1362 - Std config removal
• Issue #1361 - Parcelport plugins
• Issue #1360 - Detuplify transfer_action
• Issue #1359 - Removed obsolete checks
• Issue #1358 - Fixing 1352
• Issue #1357 - Variadic templates support for runtime_support and components
• Issue #1356 - fixed coordinate test for intel13
• Issue #1355 - fixed coordinate.hpp
• Issue #1354 - Lexicographical Compare completed
• Issue #1353 - HPX should set Boost_ADDITIONAL_VERSIONS flags
• Issue #1352 - Error: Cannot find action "" in type registry: HPX(bad_action_code)
• Issue #1351 - Variadic templates support for appliers
• Issue #1350 - Actions simplification
• Issue #1349 - Variadic when and wait functions
• Issue #1348 - Added hpx_init header to test files
• Issue #1347 - Another batch of variadic templates support

https://github.com/STEllAR-GROUP/hpx/issues/1369
https://github.com/STEllAR-GROUP/hpx/issues/1368
https://github.com/STEllAR-GROUP/hpx/issues/1367
https://github.com/STEllAR-GROUP/hpx/issues/1366
https://github.com/STEllAR-GROUP/hpx/issues/1363
https://github.com/STEllAR-GROUP/hpx/issues/1362
https://github.com/STEllAR-GROUP/hpx/issues/1361
https://github.com/STEllAR-GROUP/hpx/issues/1360
https://github.com/STEllAR-GROUP/hpx/issues/1359
https://github.com/STEllAR-GROUP/hpx/issues/1358
https://github.com/STEllAR-GROUP/hpx/issues/1357
https://github.com/STEllAR-GROUP/hpx/issues/1356
https://github.com/STEllAR-GROUP/hpx/issues/1355
https://github.com/STEllAR-GROUP/hpx/issues/1354
https://github.com/STEllAR-GROUP/hpx/issues/1353
https://github.com/STEllAR-GROUP/hpx/issues/1352
https://github.com/STEllAR-GROUP/hpx/issues/1351
https://github.com/STEllAR-GROUP/hpx/issues/1350
https://github.com/STEllAR-GROUP/hpx/issues/1349
https://github.com/STEllAR-GROUP/hpx/issues/1348
https://github.com/STEllAR-GROUP/hpx/issues/1347
• Issue #1346\(^{3946}\) - Segmented copy
• Issue #1345\(^{3947}\) - Attempting to fix hangs during shutdown
• Issue #1344\(^{3948}\) - Std config removal
• Issue #1343\(^{3949}\) - Removing various distribution policies for hpx::vector
• Issue #1342\(^{3950}\) - Inclusive scan
• Issue #1341\(^{3951}\) - Exclusive scan
• Issue #1340\(^{3952}\) - Adding `parallel::count` for distributed data structures, adding tests
• Issue #1339\(^{3953}\) - Update argument order for `transform_reduce`
• Issue #1337\(^{3954}\) - Fix dataflow to handle properly ranges of futures
• Issue #1336\(^{3955}\) - dataflow needs to hold onto futures passed to it
• Issue #1335\(^{3956}\) - Fails to compile with msve14
• Issue #1334\(^{3957}\) - Examples build problem
• Issue #1333\(^{3958}\) - Distributed transform reduce
• Issue #1332\(^{3959}\) - Variadic templates support for actions
• Issue #1331\(^{3960}\) - Some ambiguous calls of `map::erase` have been prevented by adding additional check in locality constructor.
• Issue #1330\(^{3961}\) - Defining Plain Actions does not work as described in the documentation
• Issue #1329\(^{3962}\) - Distributed vector cleanup
• Issue #1328\(^{3963}\) - Sync docs and comments with code in hello_world example
• Issue #1327\(^{3964}\) - Typos in docs
• Issue #1326\(^{3965}\) - Documentation and code diverged in Fibonacci tutorial
• Issue #1325\(^{3966}\) - Exceptions thrown during parcel handling are not handled correctly
• Issue #1324\(^{3967}\) - fixed bandwidth calculation
• Issue #1323\(^{3968}\) - `mmap()` failed to allocate thread stack due to insufficient resources

\(^{3946}\) https://github.com/STEllAR-GROUP/hpx/issues/1346
\(^{3947}\) https://github.com/STEllAR-GROUP/hpx/issues/1345
\(^{3948}\) https://github.com/STEllAR-GROUP/hpx/issues/1344
\(^{3949}\) https://github.com/STEllAR-GROUP/hpx/issues/1343
\(^{3950}\) https://github.com/STEllAR-GROUP/hpx/issues/1342
\(^{3951}\) https://github.com/STEllAR-GROUP/hpx/issues/1341
\(^{3952}\) https://github.com/STEllAR-GROUP/hpx/issues/1340
\(^{3953}\) https://github.com/STEllAR-GROUP/hpx/issues/1339
\(^{3954}\) https://github.com/STEllAR-GROUP/hpx/issues/1337
\(^{3955}\) https://github.com/STEllAR-GROUP/hpx/issues/1336
\(^{3956}\) https://github.com/STEllAR-GROUP/hpx/issues/1335
\(^{3957}\) https://github.com/STEllAR-GROUP/hpx/issues/1334
\(^{3958}\) https://github.com/STEllAR-GROUP/hpx/issues/1333
\(^{3959}\) https://github.com/STEllAR-GROUP/hpx/issues/1332
\(^{3960}\) https://github.com/STEllAR-GROUP/hpx/issues/1331
\(^{3961}\) https://github.com/STEllAR-GROUP/hpx/issues/1330
\(^{3962}\) https://github.com/STEllAR-GROUP/hpx/issues/1329
\(^{3963}\) https://github.com/STEllAR-GROUP/hpx/issues/1328
\(^{3964}\) https://github.com/STEllAR-GROUP/hpx/issues/1327
\(^{3965}\) https://github.com/STEllAR-GROUP/hpx/issues/1326
\(^{3966}\) https://github.com/STEllAR-GROUP/hpx/issues/1325
\(^{3967}\) https://github.com/STEllAR-GROUP/hpx/issues/1324
\(^{3968}\) https://github.com/STEllAR-GROUP/hpx/issues/1323
• Issue #1322^3969 - HPX fails to build aa182cf
• Issue #1321^3970 - Limiting size of outgoing messages while coalescing parcels
• Issue #1320^3971 - passing a future with launch::deferred in remote function call causes hang
• Issue #1319^3972 - An exception when tries to specify number high priority threads with abp-priority
• Issue #1318^3973 - Unable to run program with abp-priority and numa-sensitivity enabled
• Issue #1317^3974 - N4071 Search/Search_n finished, minor changes
• Issue #1316^3975 - Add config option to make -hpx.run_hpx_main!=1 the default
• Issue #1314^3976 - Variadic support for async and apply
• Issue #1313^3977 - Adjust when_any/some to the latest proposed interfaces
• Issue #1312^3978 - Fixing #857: hpx::naming::locality leaks parcelport specific information into the public interface
• Issue #1311^3979 - Distributed get/er/set/er_values for distributed vector
• Issue #1310^3980 - Crashing in hpx::parcelset::policies::mpi::connection_handler::handle_messages() on SuperMIC
• Issue #1308^3981 - Unable to execute an application with –hpx:threads
• Issue #1307^3982 - merge_graph linking issue
• Issue #1306^3983 - First batch of variadic templates support
• Issue #1305^3984 - Create a compiler wrapper
• Issue #1304^3985 - Provide a compiler wrapper for hpx
• Issue #1303^3986 - Drop support for GCC44
• Issue #1302^3987 - Fixing #1297
• Issue #1301^3988 - Compilation error when tried to use boost range iterators with wait_all
• Issue #1298^3989 - Distributed vector
• Issue #1297^3990 - Unable to invoke component actions recursively

^3969 https://github.com/STEllAR-GROUP/hpx/issues/1322
^3970 https://github.com/STEllAR-GROUP/hpx/issues/1321
^3971 https://github.com/STEllAR-GROUP/hpx/issues/1320
^3972 https://github.com/STEllAR-GROUP/hpx/issues/1319
^3973 https://github.com/STEllAR-GROUP/hpx/issues/1318
^3974 https://github.com/STEllAR-GROUP/hpx/issues/1317
^3975 https://github.com/STEllAR-GROUP/hpx/issues/1316
^3976 https://github.com/STEllAR-GROUP/hpx/issues/1314
^3977 https://github.com/STEllAR-GROUP/hpx/issues/1313
^3978 https://github.com/STEllAR-GROUP/hpx/issues/1312
^3979 https://github.com/STEllAR-GROUP/hpx/issues/1311
^3980 https://github.com/STEllAR-GROUP/hpx/issues/1310
^3981 https://github.com/STEllAR-GROUP/hpx/issues/1308
^3982 https://github.com/STEllAR-GROUP/hpx/issues/1307
^3983 https://github.com/STEllAR-GROUP/hpx/issues/1306
^3984 https://github.com/STEllAR-GROUP/hpx/issues/1305
^3985 https://github.com/STEllAR-GROUP/hpx/issues/1304
^3986 https://github.com/STEllAR-GROUP/hpx/issues/1303
^3987 https://github.com/STEllAR-GROUP/hpx/issues/1302
^3988 https://github.com/STEllAR-GROUP/hpx/issues/1301
^3989 https://github.com/STEllAR-GROUP/hpx/issues/1298
^3990 https://github.com/STEllAR-GROUP/hpx/issues/1297

Chapter 2. What’s so special about HPX?
• Issue #1294 - HDF5 build error
• Issue #1275 - The parcelport implementation is non-optimal
• Issue #1267 - Added classes and unit tests for local_file, orangefs_file and pxfs_file
• Issue #1264 - Error “assertion ‘!m_fun’ failed” randomly occurs when using TCP
• Issue #1254 - thread binding seems to not work properly
• Issue #1220 - parallel::copy_if is broken
• Issue #1217 - Find a better way of fixing the issue patched by #1216
• Issue #1168 - Starting HPX on Cray machines using aprun isn’t working correctly
• Issue #1085 - Replace startup and shutdown barriers with broadcasts
• Issue #981 - With SLURM, --hpx:threads=8 should not be necessary
• Issue #857 - hpx::naming::locality leaks parcelport specific information into the public interface
• Issue #850 - “flush” not documented
• Issue #763 - Create buildbot instance that uses std::bind as HPX_STD_BIND
• Issue #680 - Convert parcel ports into a plugin system
• Issue #582 - Make exception thrown from HPX threads available from hpx::init
• Issue #504 - Refactor Dataflow LCO to work with futures
• Issue #196 - Don’t store copies of the locality network metadata in the gva table

2.10.16 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.

3991 https://github.com/STEllAR-GROUP/hpx/issues/1294
3992 https://github.com/STEllAR-GROUP/hpx/issues/1275
3993 https://github.com/STEllAR-GROUP/hpx/issues/1267
3994 https://github.com/STEllAR-GROUP/hpx/issues/1264
3995 https://github.com/STEllAR-GROUP/hpx/issues/1254
3996 https://github.com/STEllAR-GROUP/hpx/issues/1220
3997 https://github.com/STEllAR-GROUP/hpx/issues/1217
3998 https://github.com/STEllAR-GROUP/hpx/issues/1168
3999 https://github.com/STEllAR-GROUP/hpx/issues/1085
4000 https://github.com/STEllAR-GROUP/hpx/issues/981
4001 https://github.com/STEllAR-GROUP/hpx/issues/857
4002 https://github.com/STEllAR-GROUP/hpx/issues/850
4003 https://github.com/STEllAR-GROUP/hpx/issues/763
4004 https://github.com/STEllAR-GROUP/hpx/issues/680
4005 https://github.com/STEllAR-GROUP/hpx/issues/582
4006 https://github.com/STEllAR-GROUP/hpx/issues/504
4007 https://github.com/STEllAR-GROUP/hpx/issues/196
• We rewrote and significantly changed our build system. Please have a look at the new (now generated) documentation here: *HPX build system*. 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 variables used to configure HPX* for more information).
  
  – This also fixes a long list of issues, for more information see [Issue #1204](https://github.com/STEllAR-GROUP/hpx/issues/1204).


• 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*.[4012]

### Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

• [Issue #1296](https://github.com/STEllAR-GROUP/hpx/issues/1296) - Rename make_error_future to make_exceptional_future, adjust to N4123

• [Issue #1295](https://github.com/STEllAR-GROUP/hpx/issues/1295) - building issue

• [Issue #1293](https://github.com/STEllAR-GROUP/hpx/issues/1293) - Transpose example

• [Issue #1292](https://github.com/STEllAR-GROUP/hpx/issues/1292) - Wrong abs() function used in example

• [Issue #1291](https://github.com/STEllAR-GROUP/hpx/issues/1291) - non-synchronized shift operators have been removed

• [Issue #1290](https://github.com/STEllAR-GROUP/hpx/issues/1290) - RDTSCP is defined as true for Xeon Phi build

• [Issue #1289](https://github.com/STEllAR-GROUP/hpx/issues/1289) - Fixing 1288

• [Issue #1288](https://github.com/STEllAR-GROUP/hpx/issues/1288) - Add new performance counters

• [Issue #1287](https://github.com/STEllAR-GROUP/hpx/issues/1287) - Hierarchy scheduler broken performance counters

---

4008 https://github.com/STEllAR-GROUP/hpx/issues/1204
4009 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf
4011 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
4012 http://rostam.cct.lsu.edu/
4013 https://github.com/STEllAR-GROUP/hpx/issues/1296
4014 https://github.com/STEllAR-GROUP/hpx/issues/1295
4015 https://github.com/STEllAR-GROUP/hpx/issues/1293
4016 https://github.com/STEllAR-GROUP/hpx/issues/1292
4017 https://github.com/STEllAR-GROUP/hpx/issues/1291
4018 https://github.com/STEllAR-GROUP/hpx/issues/1290
4019 https://github.com/STEllAR-GROUP/hpx/issues/1289
4020 https://github.com/STEllAR-GROUP/hpx/issues/1288
4021 https://github.com/STEllAR-GROUP/hpx/issues/1287
- Issue #1286 - Algorithm cleanup
- Issue #1285 - Broken Links in Documentation
- Issue #1284 - Uninitialized copy
- Issue #1283 - missing boost::scoped_ptr includes
- Issue #1282 - Update documentation of build options for schedulers
- Issue #1281 - reset idle rate counter
- Issue #1280 - Bug when executing on Intel MIC
- Issue #1279 - Add improved when_all/wait_all
- Issue #1278 - Implement improved when_all/wait_all
- Issue #1277 - feature request: get access to argc argv and variables_map
- Issue #1276 - Remove merging map
- Issue #1274 - Weird (wrong) string code in papi.cpp
- Issue #1273 - Sequential task execution policy
- Issue #1272 - Avoid CMake name clash for Boost.Thread library
- Issue #1271 - Updates on HPX Test Units
- Issue #1270 - hpx/util/safe_lexical_cast.hpp is added
- Issue #1269 - Added default value for “LIB” cmake variable
- Issue #1268 - Memory Counters not working
- Issue #1266 - FindHPX.cmake is not installed
- Issue #1263 - apply_remote test takes too long
- Issue #1262 - Chrono cleanup
- Issue #1261 - Need make install for papi counters and this builds all the examples
- Issue #1260 - Documentation of Stencil example claims

https://github.com/STEllAR-GROUP/hpx/issues/1286
https://github.com/STEllAR-GROUP/hpx/issues/1285
https://github.com/STEllAR-GROUP/hpx/issues/1284
https://github.com/STEllAR-GROUP/hpx/issues/1283
https://github.com/STEllAR-GROUP/hpx/issues/1282
https://github.com/STEllAR-GROUP/hpx/issues/1281
https://github.com/STEllAR-GROUP/hpx/issues/1280
https://github.com/STEllAR-GROUP/hpx/issues/1279
https://github.com/STEllAR-GROUP/hpx/issues/1278
https://github.com/STEllAR-GROUP/hpx/issues/1277
https://github.com/STEllAR-GROUP/hpx/issues/1276
https://github.com/STEllAR-GROUP/hpx/issues/1274
https://github.com/STEllAR-GROUP/hpx/issues/1273
https://github.com/STEllAR-GROUP/hpx/issues/1272
https://github.com/STEllAR-GROUP/hpx/issues/1271
https://github.com/STEllAR-GROUP/hpx/issues/1270
https://github.com/STEllAR-GROUP/hpx/issues/1269
https://github.com/STEllAR-GROUP/hpx/issues/1268
https://github.com/STEllAR-GROUP/hpx/issues/1266
https://github.com/STEllAR-GROUP/hpx/issues/1263
https://github.com/STEllAR-GROUP/hpx/issues/1262
https://github.com/STEllAR-GROUP/hpx/issues/1261
https://github.com/STEllAR-GROUP/hpx/issues/1260
• Issue #1259 - Avoid double-linking Boost on Windows
• Issue #1257 - Adding additional parameter to create_thread
• Issue #1256 - added buildbot changes to release notes
• Issue #1255 - Cannot build MiniGhost
• Issue #1253 - hpx::thread defects
• Issue #1252 - HPX_PREFIX is too fragile
• Issue #1250 - switch_to_fiber_emulation does not work properly
• Issue #1249 - Documentation is generated under Release folder
• Issue #1248 - Fix usage of hpx_generic_coroutine_context and get tests passing on powerpc
• Issue #1247 - Dynamic linking error
• Issue #1246 - Make cpuid.cpp C++11 compliant
• Issue #1245 - HPX fails on startup (setting thread affinity mask)
• Issue #1244 - HPX_WITH_RDTSC configure test fails, but should succeed
• Issue #1243 - CTest dashboard info for CSCS CDash drop location
• Issue #1242 - Mac fixes
• Issue #1241 - Failure in Distributed with Boost 1.56
• Issue #1240 - fix a race condition in examples.diskperformance
• Issue #1239 - fix wait_each in examples.diskperformance
• Issue #1238 - Fixed #1237: hpx::util::portable_binary_iarchive failed
• Issue #1237 - hpx::util::portable_binary_iarchive faileds
• Issue #1235 - Fixing clang warnings and errors
• Issue #1234 - TCP runs fail: Transport endpoint is not connected
• Issue #1233 - Making sure the correct number of threads is registered with AGAS
• Issue #1232 - Fixing race in wait_xxx
• Issue #1231 - Parallel minmax
• Issue #1230 - Distributed run of 1d_stencil_8 uses less threads than spec. & sometimes gives errors
• Issue #1229 - Unstable number of threads
• Issue #1228 - HPX link error (cmake / MPI)
• Issue #1226 - Warning about struct/class thread_counters
• Issue #1225 - Adding parallel::replace etc
• Issue #1224 - Extending dataflow to pass through non-future arguments
• Issue #1223 - Remaining find algorithms implemented, N4071
• Issue #1222 - Merging all the changes
• Issue #1221 - No error output when using mpirun with hpx
• Issue #1219 - Adding new AGAS cache performance counters
• Issue #1216 - Fixing using futures (clients) as arguments to actions
• Issue #1215 - Error compiling simple component
• Issue #1214 - Stencil docs
• Issue #1213 - Using more than a few dozen MPI processes on SuperMike results in a seg fault before getting to hpx_main
• Issue #1212 - Parallel rotate
• Issue #1211 - Direct actions cause the future’s shared_state to be leaked
• Issue #1210 - Refactored local::promise to be standard conformant
• Issue #1209 - Improve command line handling
• Issue #1208 - Adding parallel::reverse and parallel::reverse_copy
• Issue #1207 - Add copy_backward and move_backward
• Issue #1206 - N4071 additional algorithms implemented

https://github.com/STEllAR-GROUP/hpx/issues/1232
https://github.com/STEllAR-GROUP/hpx/issues/1231
https://github.com/STEllAR-GROUP/hpx/issues/1230
https://github.com/STEllAR-GROUP/hpx/issues/1229
https://github.com/STEllAR-GROUP/hpx/issues/1228
https://github.com/STEllAR-GROUP/hpx/issues/1226
https://github.com/STEllAR-GROUP/hpx/issues/1225
https://github.com/STEllAR-GROUP/hpx/issues/1224
https://github.com/STEllAR-GROUP/hpx/issues/1223
https://github.com/STEllAR-GROUP/hpx/issues/1222
https://github.com/STEllAR-GROUP/hpx/issues/1221
https://github.com/STEllAR-GROUP/hpx/issues/1219
https://github.com/STEllAR-GROUP/hpx/issues/1216
https://github.com/STEllAR-GROUP/hpx/issues/1215
https://github.com/STEllAR-GROUP/hpx/issues/1214
https://github.com/STEllAR-GROUP/hpx/issues/1213
https://github.com/STEllAR-GROUP/hpx/issues/1212
https://github.com/STEllAR-GROUP/hpx/issues/1211
https://github.com/STEllAR-GROUP/hpx/issues/1210
https://github.com/STEllAR-GROUP/hpx/issues/1209
https://github.com/STEllAR-GROUP/hpx/issues/1208
https://github.com/STEllAR-GROUP/hpx/issues/1207
https://github.com/STEllAR-GROUP/hpx/issues/1206
• Issue #1204 — Cmake simplification and various other minor changes
• Issue #1203 — Implementing new launch policy for (local) async: hpx::launch::fork.
• Issue #1202 — Failed assertion in connection_cache.hpp
• Issue #1201 — pkg-config doesn’t add mpi link directories
• Issue #1200 — Error when querying time performance counters
• Issue #1199 — library path is now configurable (again)
• Issue #1198 — Error when querying performance counters
• Issue #1197 — tests fail with intel compiler
• Issue #1196 — Silence several warnings
• Issue #1195 — Rephrase initializers to work with VC++ 2012
• Issue #1194 — Simplify parallel algorithms
• Issue #1193 — Adding parallel::equal
• Issue #1192 — HPX(out_of_memory) on including <hpx/hpx.hpp>
• Issue #1191 — Fixing #1189
• Issue #1190 — Chrono cleanup
• Issue #1189 — Deadlock .. somewhere? (probably serialization)
• Issue #1188 — Removed future::get_status()
• Issue #1186 — Fixed FindOpenCL to find current AMD APP SDK
• Issue #1184 — Tweaking future unwrapping
• Issue #1183 — Extended parallel::reduce
• Issue #1182 — future::unwrap hangs for launch::deferred
• Issue #1181 — Adding all_of, any_of, and none_of and corresponding documentation
• Issue #1180 — hpx::cout defect

https://github.com/STEllAR-GROUP/hpx/issues/1204
https://github.com/STEllAR-GROUP/hpx/issues/1203
https://github.com/STEllAR-GROUP/hpx/issues/1202
https://github.com/STEllAR-GROUP/hpx/issues/1201
https://github.com/STEllAR-GROUP/hpx/issues/1200
https://github.com/STEllAR-GROUP/hpx/issues/1199
https://github.com/STEllAR-GROUP/hpx/issues/1198
https://github.com/STEllAR-GROUP/hpx/issues/1197
https://github.com/STEllAR-GROUP/hpx/issues/1196
https://github.com/STEllAR-GROUP/hpx/issues/1195
https://github.com/STEllAR-GROUP/hpx/issues/1194
https://github.com/STEllAR-GROUP/hpx/issues/1193
https://github.com/STEllAR-GROUP/hpx/issues/1192
https://github.com/STEllAR-GROUP/hpx/issues/1191
https://github.com/STEllAR-GROUP/hpx/issues/1190
https://github.com/STEllAR-GROUP/hpx/issues/1189
https://github.com/STEllAR-GROUP/hpx/issues/1188
https://github.com/STEllAR-GROUP/hpx/issues/1186
https://github.com/STEllAR-GROUP/hpx/issues/1184
https://github.com/STEllAR-GROUP/hpx/issues/1183
https://github.com/STEllAR-GROUP/hpx/issues/1182
https://github.com/STEllAR-GROUP/hpx/issues/1181
https://github.com/STEllAR-GROUP/hpx/issues/1180

Chapter 2. What’s so special about HPX?
• Issue #1179 - `hpx::async` does not work for member function pointers when called on types with self-defined unary operator*

• Issue #1178 - Implemented variadic `hpx::util::zip_iterator`

• Issue #1177 - MPI parcelport defect

• Issue #1176 - `HPX_DEFINE_COMPONENT_CONST_ACTION_TPL` does not have a 2-argument version

• Issue #1175 - Create `util::zip_iterator` working with `util::tuple<>`

• Issue #1174 - Error Building HPX on linux, `root_certificate_authority.cpp`

• Issue #1173 - `hpx::cout` output lost

• Issue #1172 - HPX build error with Clang 3.4.2

• Issue #1171 - `CMAKE_INSTALL_PREFIX` ignored

• Issue #1170 - Close `hpx_benchmarks` repository on Github

• Issue #1169 - Buildbot emails have syntax error in url

• Issue #1168 - Merge partial implementation of standards proposal N3960

• Issue #1167 - Fixed several compiler warnings

• Issue #1165 - `cmake` warns: “tests.regressions.actions” does not exist

• Issue #1164 - Want my own serialization of `hpx::future`

• Issue #1162 - Segfault in `hello_world` example

• Issue #1161 - Use `HPX_ASSERT` to aid the compiler

• Issue #1160 - Do not put `-DNDEBUG` into `hpx_application.pc`

• Issue #1159 - Support Clang 3.4.2

• Issue #1158 - Fixed #1157: Rename `when_n/wait_n`, add `when_xxx_n/wait_xxx_n`

• Issue #1157 - Rename `when_n/wait_n`, add `when_xxx_n/wait_xxx_n`

• Issue #1156 - Force inlining fails

• Issue #1155 - changed header of printout to be compatible with python csv module

---

4114 https://github.com/STEllAR-GROUP/hpx/issues/1179
4115 https://github.com/STEllAR-GROUP/hpx/issues/1178
4116 https://github.com/STEllAR-GROUP/hpx/issues/1177
4117 https://github.com/STEllAR-GROUP/hpx/issues/1176
4118 https://github.com/STEllAR-GROUP/hpx/issues/1175
4119 https://github.com/STEllAR-GROUP/hpx/issues/1174
4120 https://github.com/STEllAR-GROUP/hpx/issues/1173
4121 https://github.com/STEllAR-GROUP/hpx/issues/1172
4122 https://github.com/STEllAR-GROUP/hpx/issues/1171
4123 https://github.com/STEllAR-GROUP/hpx/issues/1170
4124 https://github.com/STEllAR-GROUP/hpx/issues/1169
4125 https://github.com/STEllAR-GROUP/hpx/issues/1167
4126 https://github.com/STEllAR-GROUP/hpx/issues/1166
4127 https://github.com/STEllAR-GROUP/hpx/issues/1165
4128 https://github.com/STEllAR-GROUP/hpx/issues/1164
4129 https://github.com/STEllAR-GROUP/hpx/issues/1162
4130 https://github.com/STEllAR-GROUP/hpx/issues/1161
4131 https://github.com/STEllAR-GROUP/hpx/issues/1159
4132 https://github.com/STEllAR-GROUP/hpx/issues/1158
4133 https://github.com/STEllAR-GROUP/hpx/issues/1157
4134 https://github.com/STEllAR-GROUP/hpx/issues/1156
4135 https://github.com/STEllAR-GROUP/hpx/issues/1155
4136 https://github.com/STEllAR-GROUP/hpx/issues/1155
• Issue #1154 - Fixing iostreams
• Issue #1153 - Standard manipulators (like std::endl) do not work with hpx::ostream
• Issue #1152 - Functions revamp
• Issue #1151 - Suppressing cmake 3.0 policy warning for CMP0026
• Issue #1150 - Client Serialization error
• Issue #1149 - Segfault on Stampede
• Issue #1148 - Refactoring mini-ghost
• Issue #1147 - N3960 copy_if and copy_n implemented and tested
• Issue #1146 - Stencil print
• Issue #1145 - N3960 hpx::parallel::copy implemented and tested
• Issue #1144 - OpenMP examples 1d_stencil do not build
• Issue #1143 - 1d_stencil OpenMP examples do not build
• Issue #1142 - Cannot build HPX with gcc 4.6 on OS X
• Issue #1140 - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
• Issue #1139 - hpx/hpx/config/compiler_specific.hpp
• Issue #1138 - clean up pkg-config files
• Issue #1137 - Improvements to create binary packages
• Issue #1136 - HPX_GCC_VERSION not defined on all compilers
• Issue #1135 - Avoiding collision between winsock2.h and windows.h
• Issue #1134 - Making sure, that hpx::finalize can be called from any locality
• Issue #1133 - 1d stencil examples
• Issue #1131 - Refactor unique_function implementation
• Issue #1130 - Unique function

4137 https://github.com/STEllAR-GROUP/hpx/issues/1154
4138 https://github.com/STEllAR-GROUP/hpx/issues/1153
4139 https://github.com/STEllAR-GROUP/hpx/issues/1152
4140 https://github.com/STEllAR-GROUP/hpx/issues/1151
4141 https://github.com/STEllAR-GROUP/hpx/issues/1150
4142 https://github.com/STEllAR-GROUP/hpx/issues/1149
4143 https://github.com/STEllAR-GROUP/hpx/issues/1148
4144 https://github.com/STEllAR-GROUP/hpx/issues/1147
4145 https://github.com/STEllAR-GROUP/hpx/issues/1146
4146 https://github.com/STEllAR-GROUP/hpx/issues/1145
4147 https://github.com/STEllAR-GROUP/hpx/issues/1144
4148 https://github.com/STEllAR-GROUP/hpx/issues/1143
4149 https://github.com/STEllAR-GROUP/hpx/issues/1142
4150 https://github.com/STEllAR-GROUP/hpx/issues/1140
4151 https://github.com/STEllAR-GROUP/hpx/issues/1139
4152 https://github.com/STEllAR-GROUP/hpx/issues/1138
4153 https://github.com/STEllAR-GROUP/hpx/issues/1137
4154 https://github.com/STEllAR-GROUP/hpx/issues/1136
4155 https://github.com/STEllAR-GROUP/hpx/issues/1135
4156 https://github.com/STEllAR-GROUP/hpx/issues/1134
4157 https://github.com/STEllAR-GROUP/hpx/issues/1133
4158 https://github.com/STEllAR-GROUP/hpx/issues/1131
4159 https://github.com/STEllAR-GROUP/hpx/issues/1130
• Issue #1129 - Some fixes to the Build system on OS X
• Issue #1128 - Action future args
• Issue #1127 - Executor causes segmentation fault
• Issue #1124 - Adding new API functions: register_id_with_basename, unregister_id_with_basename, find_ids_from_basename; adding test
• Issue #1123 - Reduce nesting of try-catch construct in encode_parcel?
• Issue #1122 - Client base fixes
• Issue #1121 - Update hpxrun.py.in
• Issue #1120 - HTTS2 tests compile errors on v110 (VS2012)
• Issue #1119 - Remove references to boost::atomic in accumulator example
• Issue #1118 - Only build test thread_pool_executor_1114_test if HPX_LOCAL_SCHEDULER is set
• Issue #1117 - local_queue_executor linker error on vc110
• Issue #1116 - Disabled performance counter should give runtime errors, not invalid data
• Issue #1115 - Compile error with Intel C++ 13.1
• Issue #1114 - Default constructed executor is not usable
• Issue #1113 - Fast compilation of logging causes ABI incompatibilities between different NDEBUG values
• Issue #1112 - Using thread_pool_executors causes segfault
• Issue #1111 - hpx::threads::get_thread_data always returns zero
• Issue #1110 - Remove unnecessary null pointer checks
• Issue #1109 - More tests adjustments
• Issue #1108 - Clarify build rules for “libboost_atomic-mt.so”?
• Issue #1107 - Remove unnecessary null pointer checks
• Issue #1106 - network_storage benchmark improvements, adding legends to plots and tidying layout
• Issue #1105 - Add more plot outputs and improve instructions doc
• Issue #1104 - Complete quoting for parameters of some CMake commands
• Issue #1103 - Work on test/scripts
• Issue #1102 - Changed minimum requirement of window install to 2012
• Issue #1101 - Changed minimum requirement of window install to 2012
• Issue #1100 - Changed readme to no longer specify using MSVC 2010 compiler
• Issue #1099 - Error returning futures from component actions
• Issue #1098 - Improve storage test
• Issue #1097 - data_actions quickstart example calls missing function decorate_action of data_get_action
• Issue #1096 - MPI parcelport broken with new zero copy optimization
• Issue #1095 - Warning C4005: _WIN32_WINNT: Macro redefinition
• Issue #1094 - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
• Issue #1093 - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS
• Issue #1092 - Rename unique_future<> back to future<>
• Issue #1091 - Inconsistent error message
• Issue #1090 - On windows 8.1 the examples crashed if using more than one os thread
• Issue #1089 - Components should be allowed to have their own executor
• Issue #1088 - Add possibility to select a network interface for the ibverbs parcelport
• Issue #1087 - ibverbs and ipc parcelport uses zero copy optimization
• Issue #1083 - Make shell examples copyable in docs
• Issue #1082 - Implement proper termination detection during shutdown
• Issue #1081 - Implement thread_specific_ptr for hpx::threads
• Issue #1072 - make install not working properly
• Issue #1070 - Complete quoting for parameters of some CMake commands

https://github.com/STEllAR-GROUP/hpx/issues/1104
https://github.com/STEllAR-GROUP/hpx/issues/1103
https://github.com/STEllAR-GROUP/hpx/issues/1102
https://github.com/STEllAR-GROUP/hpx/issues/1101
https://github.com/STEllAR-GROUP/hpx/issues/1100
https://github.com/STEllAR-GROUP/hpx/issues/1099
https://github.com/STEllAR-GROUP/hpx/issues/1098
https://github.com/STEllAR-GROUP/hpx/issues/1097
https://github.com/STEllAR-GROUP/hpx/issues/1096
https://github.com/STEllAR-GROUP/hpx/issues/1095
https://github.com/STEllAR-GROUP/hpx/issues/1094
https://github.com/STEllAR-GROUP/hpx/issues/1093
https://github.com/STEllAR-GROUP/hpx/issues/1092
https://github.com/STEllAR-GROUP/hpx/issues/1091
https://github.com/STEllAR-GROUP/hpx/issues/1090
https://github.com/STEllAR-GROUP/hpx/issues/1089
https://github.com/STEllAR-GROUP/hpx/issues/1088
https://github.com/STEllAR-GROUP/hpx/issues/1087
https://github.com/STEllAR-GROUP/hpx/issues/1083
https://github.com/STEllAR-GROUP/hpx/issues/1082
https://github.com/STEllAR-GROUP/hpx/issues/1081
https://github.com/STEllAR-GROUP/hpx/issues/1072
https://github.com/STEllAR-GROUP/hpx/issues/1070
• Issue #1059 - Fix more unused variable warnings
• Issue #1051 - Implement when_each
• Issue #973 - Would like option to report hwloc bindings
• Issue #970 - Bad flags for Fortran compiler
• Issue #941 - Create a proper user level context switching class for BG/Q
• Issue #935 - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
• Issue #934 - Want to build HPX without dynamic libraries
• Issue #927 - Make hpx/lcos/reduce.hpp accept futures of id_type
• Issue #926 - All unit tests that are run with more than one thread with CTest/hpx_run_tests should configure hpx.os_threads
• Issue #925 - regression_dataflow_791 needs to be brought in line with HPX standards
• Issue #899 - Fix race conditions in regression tests
• Issue #879 - Hung test leads to cascading test failure; make tests should support the MPI parcelport
• Issue #865 - future<T> and friends shall work for movable only Ts
• Issue #847 - Dynamic libraries are not installed on OS X
• Issue #816 - First Program tutorial pull request
• Issue #799 - Wrap lexical_cast to avoid exceptions
• Issue #720 - broken configuration when using ccmake on Ubuntu
• Issue #622 - --hpx:hpx and --hpx:debug-hpx-log is nonsensical
• Issue #525 - Extend barrier LCO test to run in distributed
• Issue #515 - Multi-destination version of hpx::apply is broken
• Issue #509 - Push Boost.Atomic changes upstream
• Issue #503 - Running HPX applications on Windows should not require setting %PATH%
• Issue #461 - Add a compilation sanity test

https://github.com/STEllAR-GROUP/hpx/issues/1059
https://github.com/STEllAR-GROUP/hpx/issues/1051
https://github.com/STEllAR-GROUP/hpx/issues/973
https://github.com/STEllAR-GROUP/hpx/issues/941
https://github.com/STEllAR-GROUP/hpx/issues/935
https://github.com/STEllAR-GROUP/hpx/issues/934
https://github.com/STEllAR-GROUP/hpx/issues/927
https://github.com/STEllAR-GROUP/hpx/issues/926
https://github.com/STEllAR-GROUP/hpx/issues/925
https://github.com/STEllAR-GROUP/hpx/issues/899
https://github.com/STEllAR-GROUP/hpx/issues/879
https://github.com/STEllAR-GROUP/hpx/issues/865
https://github.com/STEllAR-GROUP/hpx/issues/847
https://github.com/STEllAR-GROUP/hpx/issues/816
https://github.com/STEllAR-GROUP/hpx/issues/799
https://github.com/STEllAR-GROUP/hpx/issues/720
https://github.com/STEllAR-GROUP/hpx/issues/622
https://github.com/STEllAR-GROUP/hpx/issues/525
https://github.com/STEllAR-GROUP/hpx/issues/515
https://github.com/STEllAR-GROUP/hpx/issues/509
https://github.com/STEllAR-GROUP/hpx/issues/503
https://github.com/STEllAR-GROUP/hpx/issues/461
• Issue #456[^4229] - hpx_run_tests.py should log output from tests that timeout
• Issue #454[^4230] - Investigate threadmanager performance
• Issue #345[^4231] - Add more versatile environmental/cmake variable support to hpx_find_* CMake macros
• Issue #209[^4232] - Support multiple configurations in generated build files
• Issue #190[^4233] - hpx::cout should be a std::ostream
• Issue #189[^4234] - iostreams component should use startup/shutdown functions
• Issue #183[^4235] - Use Boost.ICL for correctness in AGAS
• Issue #44[^4236] - Implement real futures

### 2.10.17 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.

[^4229]: https://github.com/STEllAR-GROUP/hpx/issues/456
[^4230]: https://github.com/STEllAR-GROUP/hpx/issues/454
[^4231]: https://github.com/STEllAR-GROUP/hpx/issues/345
[^4232]: https://github.com/STEllAR-GROUP/hpx/issues/209
[^4233]: https://github.com/STEllAR-GROUP/hpx/issues/190
[^4234]: https://github.com/STEllAR-GROUP/hpx/issues/189
[^4235]: https://github.com/STEllAR-GROUP/hpx/issues/183
[^4236]: https://github.com/STEllAR-GROUP/hpx/issues/44
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. 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. 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.

• 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.

• 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.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

• Issue #1086 - Expose internal boost::shared_array to allow user management of array lifetime
• Issue #1083 - Make shell examples copyable in docs
• Issue #1080 - /threads{locality#*/total}/count/cumulative broken
• Issue #1079 - Build problems on OS X
• Issue #1078 - Improve robustness of component loading
• Issue #1077 - Fix a missing enum definition for ‘take’ mode
• Issue #1076 - Merge Jb master
• Issue #1075 - Unknown CMake command “add_hpx_pseudo_target”
• Issue #1074 - Implement `apply_continue_callback` and `apply_colocated_callback`
• Issue #1073 - The new `apply_colocated` and `async_colocated` functions lead to automatic registered functions
• Issue #1071 - Remove deferred_packaged_task
• Issue #1069 - serialize_buffer with allocator fails at destruction
• Issue #1068 - Coroutine include and forward declarations missing
• Issue #1067 - Add allocator support to `util::serialize_buffer`
• Issue #1066 - Allow for MPI_Init being called before HPX launches
• Issue #1065 - AGAS cache isn’t used/populated on worker localities
• Issue #1064 - Reorder includes to ensure ws2 includes early
• Issue #1063 - Add `hpx::runtime::suspend` and `hpx::runtime::resume`
• Issue #1062 - Fix `async_continue` to properly handle return types
• Issue #1061 - Implement `async_colocated` and `apply_colocated`
• Issue #1060 - Implement minimal component migration
• Issue #1058 - Remove `HPX_UTIL_TUPLE` from code base
• Issue #1057 - Add performance counters for threading subsystem
• Issue #1055 - Thread allocation uses two memory pools
• Issue #1053 - Work stealing flawed
• Issue #1052 - Fix a number of warnings
• Issue #1049 - Fixes for TLS on OSX and more reliable test running
• Issue #1048 - Fixing after 588 hang
• Issue #1047 - Use port ‘0’ for networking when using one locality
• Issue #1046 - composable_guard test is broken when having more than one thread
• Issue #1045 - Security missing headers
• Issue #1044 - Native TLS on FreeBSD via __thread
• Issue #1043 - async et.al. compute the wrong result type
• Issue #1042 - async et.al. implicitly unwrap reference_wrappers
• Issue #1041 - Remove redundant costly Kleene stars from regex searches
• Issue #1040 - CMake script regex match patterns has unnecessary kleenes
• Issue #1039 - Remove use of Boost.Move and replace with std::move and real rvalue refs
• Issue #1038 - Bump minimal required Boost to 1.49.0
• Issue #1037 - Implicit unwrapping of futures in async broken
• Issue #1036 - Scheduler hangs when user code attempts to “block” OS-threads
• Issue #1035 - Idle-rate counter always reports 100% idle rate
• Issue #1034 - Symbolic name registration causes application hangs
• Issue #1033 - Application options read in from an options file generate an error message
• Issue #1032 - hpx::id_type local reference counting is wrong
• Issue #1031 - Negative entry in reference count table
• Issue #1030 - Implement condition_variable
• Issue #1029 - Deadlock in thread scheduling subsystem
• Issue #1028 - HPX-thread cumulative count performance counters report incorrect value
• Issue #1027 - Expose hpx::thread_interrupted error code as a separate exception type
• Issue #1026 - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
• Issue #1025 - future::wait_for/wait_until do not remove callback
• Issue #1024 - Remove dependence to boost assert and create hpx assert
• Issue #1023 - Segfaults with tmalloc
• Issue #1022 - prerequisites link in readme is broken
• Issue #1020 - HPX Deadlock on external synchronization
• Issue #1019 - Convert using BOOST_ASSERT to HPX_ASSERT
• Issue #1018 - compiling bug with gcc 4.8.1
• Issue #1017 - Possible crash in io_pool executor
• Issue #1016 - Crash at startup
• Issue #1014 - Implement Increment/Decrement Merging
• Issue #1013 - Add more logging channels to enable greater control over logging granularity
• Issue #1012 - crash on --hpx:debug-hpx-log and --hpx:debug-agas-log lead to non-thread safe writes
• Issue #1011 - After installation, running applications from the build/staging directory no longer works
• Issue #1010 - Mergeable decrement requests are not being merged
• Issue #1009 - --hpx:list-symbolic-names crashes
• Issue #1007 - Components are not properly destroyed
• Issue #1006 - Segfault/hang in set_data
• Issue #1003 - Performance counter naming issue
• Issue #982 - Race condition during startup
• Issue #912 - OS X: component type not found in map
• Issue #663 - Create a buildbot slave based on Clang 3.2/OSX
• Issue #636 - Expose this_locality::apply<act>(p1, p2); for local execution

https://github.com/STEllAR-GROUP/hpx/issues/1026
https://github.com/STEllAR-GROUP/hpx/issues/1025
https://github.com/STEllAR-GROUP/hpx/issues/1024
https://github.com/STEllAR-GROUP/hpx/issues/1023
https://github.com/STEllAR-GROUP/hpx/issues/1022
https://github.com/STEllAR-GROUP/hpx/issues/1020
https://github.com/STEllAR-GROUP/hpx/issues/1019
https://github.com/STEllAR-GROUP/hpx/issues/1018
https://github.com/STEllAR-GROUP/hpx/issues/1017
https://github.com/STEllAR-GROUP/hpx/issues/1016
https://github.com/STEllAR-GROUP/hpx/issues/1014
https://github.com/STEllAR-GROUP/hpx/issues/1013
https://github.com/STEllAR-GROUP/hpx/issues/1012
https://github.com/STEllAR-GROUP/hpx/issues/1011
https://github.com/STEllAR-GROUP/hpx/issues/1010
https://github.com/STEllAR-GROUP/hpx/issues/1009
https://github.com/STEllAR-GROUP/hpx/issues/1007
https://github.com/STEllAR-GROUP/hpx/issues/1006
https://github.com/STEllAR-GROUP/hpx/issues/1003
https://github.com/STEllAR-GROUP/hpx/issues/982
https://github.com/STEllAR-GROUP/hpx/issues/912
https://github.com/STEllAR-GROUP/hpx/issues/663
https://github.com/STEllAR-GROUP/hpx/issues/636

Chapter 2. What’s so special about HPX?
• Issue #197 - Add --console=address option for PBS runs

• Issue #175 - Asynchronous AGAS API

2.10.18 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

• 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 - Allow one to disable array optimizations and zero copy optimizations for each parcelport

• Issue #1004 - Generate new HPX logo image for the docs

• Issue #1002 - If MPI parcelport is not available, running HPX under mpirun should fail

• Issue #1001 - Zero copy serialization raises assert

4313 https://github.com/STEllAR-GROUP/hpx/issues/197
4314 https://github.com/STEllAR-GROUP/hpx/issues/175
4315 https://github.com/STEllAR-GROUP/hpx/issues/1005
4316 https://github.com/STEllAR-GROUP/hpx/issues/1004
4317 https://github.com/STEllAR-GROUP/hpx/issues/1002
4318 https://github.com/STEllAR-GROUP/hpx/issues/1001
• Issue #1000 - Can’t connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
• Issue #999 - Optimize hpx::when_n
• Issue #998 - Fixed const-correctness
• Issue #997 - Making serialize_buffer::data() type save
• Issue #996 - Memory leak in hpx::lcos::promise
• Issue #995 - Race while registering pre-shutdown functions
• Issue #994 - thread_rescheduling regression test does not compile
• Issue #992 - Correct comments and messages
• Issue #991 - setcap cap_sys_rawio=ep for power profiling causes an HPX application to abort
• Issue #989 - Jacobi hangs during execution
• Issue #988 - multiple_init test is failing
• Issue #986 - Can’t call a function called “init” from “main” when using <hpx/hpx_main.hpp>
• Issue #984 - Reference counting tests are failing
• Issue #983 - thread_suspension_executor test fails
• Issue #980 - Terminating HPX threads don’t leave stack in virgin state
• Issue #979 - Static scheduler not in documents
• Issue #978 - Preprocessing limits are broken
• Issue #977 - Make tests.regressions.lcos.future_hang_on_get shorter
• Issue #976 - Wrong library order in pkgconfig
• Issue #975 - Please reopen #963
• Issue #974 - Option pu-offset ignored in fixing_588 branch
• Issue #972 - Cannot use MKL with HPX
• Issue #969 - Non-existent INI files requested on the command line via --hpx:config do not cause warn-
ings or errors.

- **Issue #968** - Cannot build examples in fixing_588 branch
- **Issue #967** - Command line description of `--hpx:queueing` seems wrong
- **Issue #966** - `--hpx:print-bind` physical core numbers are wrong
- **Issue #965** - Deadlock when building in Release mode
- **Issue #963** - Not all worker threads are working
- **Issue #962** - Problem with SLURM integration
- **Issue #961** - `--hpx:print-bind` outputs incorrect information
- **Issue #960** - Fix cut and paste error in documentation of `get_thread_priority`
- **Issue #959** - Change link to boost.atomic in documentation to point to boost.org
- **Issue #958** - Undefined reference to `intrusive_ptr_release`
- **Issue #957** - Make tuple standard compliant
- **Issue #956** - Segfault with a3382fb
- **Issue #955** - `--hpx:nodes` and `--hpx:nodefiles` do not work with foreign nodes
- **Issue #954** - Make order of arguments for `hpx::async` and `hpx::broadcast` consistent
- **Issue #953** - Cannot use MKL with HPX
- **Issue #952** - `register_[pre_]shutdown_function` never throw
- **Issue #951** - Assert when number of threads is greater than hardware concurrency
- **Issue #948** - `HPX_HAVE_GENERIC_CONTEXT_COROUTINES` conflicts with `HPX_HAVE_FIBER_BASED_COROUTINES`
- **Issue #947** - Need `MPI_THREAD_MULTIPLE` for backward compatibility
- **Issue #946** - HPX does not call `MPI_Finalize`
- **Issue #945** - Segfault with `hpx::lcos::broadcast`
- **Issue #944** - OS X: assertion `pu_offset_ < hardware_concurrency` failed
• Issue #943 - #include <hpx/hpx_main.hpp> does not work
• Issue #942 - Make the BG/Q work with -O3
• Issue #940 - Use separator when concatenating locality name
• Issue #939 - Refactor MPI parcelport to use MPI_Wait instead of multiple MPI_Test calls
• Issue #938 - Want to officially access client_base::gid_
• Issue #937 - client_base::gid should be private
• Issue #936 - Want doxygen-like source code index
• Issue #935 - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
• Issue #933 - Cannot build HPX with Boost 1.54.0
• Issue #932 - Components are destructed too early
• Issue #931 - Make HPX work on BG/Q
• Issue #930 - make git-docs is broken
• Issue #929 - Generating index in docs broken
• Issue #928 - Optimize hpx::util::static_ for C++11 compilers supporting magic statics
• Issue #924 - Make kill_process_tree (in process.py) more robust on Mac OSX
• Issue #923 - Correct BLAS and RNPL cmake tests
• Issue #922 - Cannot link against BLAS
• Issue #921 - Implement hpx::mem_fn
• Issue #920 - Output locality with --hpx:print-bind
• Issue #919 - Correct grammar; simplify boolean expressions
• Issue #918 - Link to hello_world.cpp is broken
• Issue #917 - adapt cmake file to new boostbook version
• Issue #916 - fix problem building documentation with xsltproc >= 1.1.27

https://github.com/STEllAR-GROUP/hpx/issues/943
https://github.com/STEllAR-GROUP/hpx/issues/942
https://github.com/STEllAR-GROUP/hpx/issues/940
https://github.com/STEllAR-GROUP/hpx/issues/939
https://github.com/STEllAR-GROUP/hpx/issues/938
https://github.com/STEllAR-GROUP/hpx/issues/937
https://github.com/STEllAR-GROUP/hpx/issues/936
https://github.com/STEllAR-GROUP/hpx/issues/935
https://github.com/STEllAR-GROUP/hpx/issues/933
https://github.com/STEllAR-GROUP/hpx/issues/932
https://github.com/STEllAR-GROUP/hpx/issues/931
https://github.com/STEllAR-GROUP/hpx/issues/930
https://github.com/STEllAR-GROUP/hpx/issues/929
https://github.com/STEllAR-GROUP/hpx/issues/928
https://github.com/STEllAR-GROUP/hpx/issues/924
https://github.com/STEllAR-GROUP/hpx/issues/923
https://github.com/STEllAR-GROUP/hpx/issues/922
https://github.com/STEllAR-GROUP/hpx/issues/921
https://github.com/STEllAR-GROUP/hpx/issues/919
https://github.com/STEllAR-GROUP/hpx/issues/918
https://github.com/STEllAR-GROUP/hpx/issues/917
https://github.com/STEllAR-GROUP/hpx/issues/916
• Issue #9154387 - Add another TBBMalloc library search path
• Issue #9144388 - Build problem with Intel compiler on Stampede (TACC)
• Issue #9134389 - fix error messages in fibonacci examples
• Issue #9114390 - Update OS X build instructions
• Issue #9104391 - Want like to specify MPI_ROOT instead of compiler wrapper script
• Issue #9094392 - Warning about void* arithmetic
• Issue #9084393 - Buildbot for MIC is broken
• Issue #9064394 - Can’t use --hpx:bind=balanced with multiple MPI processes
• Issue #9054395 - --hpx:bind documentation should describe full grammar
• Issue #9044396 - Add hpx::lcos::fold and hpx::lcos::inverse_fold collective operation
• Issue #9034397 - Add hpx::when_any_swapped()
• Issue #9024398 - Add hpx::lcos::reduce collective operation
• Issue #9014399 - Web documentation is not searchable
• Issue #9004400 - Web documentation for trunk has no index
• Issue #8984401 - Some tests fail with GCC 4.8.1 and MPI parcel port
• Issue #8974402 - HWLOC causes failures on Mac
• Issue #8964403 - pu-offset leads to startup error
• Issue #8954404 - hpx::get_locality_name not defined
• Issue #8944405 - Race condition at shutdown
• Issue #8934406 - --hpx:print-bind switches std::cout to hexadecimal mode
• Issue #8924407 - hwloc_topology_load can be expensive – don’t call multiple times
• Issue #8914408 - The documentation for get_locality_name is wrong
• Issue #8904409 - --hpx:print-bind should not exit

https://github.com/STEllAR-GROUP/hpx/issues/915
https://github.com/STEllAR-GROUP/hpx/issues/914
https://github.com/STEllAR-GROUP/hpx/issues/913
https://github.com/STEllAR-GROUP/hpx/issues/911
https://github.com/STEllAR-GROUP/hpx/issues/910
https://github.com/STEllAR-GROUP/hpx/issues/909
https://github.com/STEllAR-GROUP/hpx/issues/908
https://github.com/STEllAR-GROUP/hpx/issues/906
https://github.com/STEllAR-GROUP/hpx/issues/905
https://github.com/STEllAR-GROUP/hpx/issues/904
https://github.com/STEllAR-GROUP/hpx/issues/903
https://github.com/STEllAR-GROUP/hpx/issues/902
https://github.com/STEllAR-GROUP/hpx/issues/901
https://github.com/STEllAR-GROUP/hpx/issues/900
https://github.com/STEllAR-GROUP/hpx/issues/898
https://github.com/STEllAR-GROUP/hpx/issues/897
https://github.com/STEllAR-GROUP/hpx/issues/896
https://github.com/STEllAR-GROUP/hpx/issues/895
https://github.com/STEllAR-GROUP/hpx/issues/894
https://github.com/STEllAR-GROUP/hpx/issues/893
https://github.com/STEllAR-GROUP/hpx/issues/892
https://github.com/STEllAR-GROUP/hpx/issues/891
https://github.com/STEllAR-GROUP/hpx/issues/890

2.10. Releases
• Issue #889 - --hpx:debug-hpx-log=FILE does not work
• Issue #888 - MPI parcelport does not exit cleanly for –hpx:print-bind
• Issue #887 - Choose thread affinities more cleverly
• Issue #886 - Logging documentation is confusing
• Issue #885 - Two threads are slower than one
• Issue #884 - is_callable failing with member pointers in C++11
• Issue #883 - Need help with is_callable_test
• Issue #882 - tests.regressions.lcos.future_hang_on_get does not terminate
• Issue #881 - tests/regressions/block_matrix/matrix.hh won’t compile with GCC 4.8.1
• Issue #880 - HPX does not work on OS X
• Issue #878 - future::unwrap triggers assertion
• Issue #877 - “make tests” has build errors on Ubuntu 12.10
• Issue #876 - tcmalloc is used by default, even if it is not present
• Issue #875 - global_fixture is defined in a header file
• Issue #874 - Some tests take very long
• Issue #873 - Add block-matrix code as regression test
• Issue #872 - HPX documentation does not say how to run tests with detailed output
• Issue #871 - All tests fail with “make test”
• Issue #870 - Please explicitly disable serialization in classes that don’t support it
• Issue #869 - boost_any test failing
• Issue #867 - Reduce the number of copies of hpx::function arguments
• Issue #863 - Futures should not require a default constructor
• Issue #862 - value_or_error shall not default construct its result
- Issue #861 - HPX_UNUSED macro
- Issue #860 - Add functionality to copy construct a component
- Issue #859 - hpx::endl should flush
- Issue #858 - Create hpx::get_ptr<> allowing to access component implementation
- Issue #855 - Implement hpx::INVOC
- Issue #854 - hpx/hpx.hpp does not include hpx/include/iostreams.hpp
- Issue #853 - Feature request: null future
- Issue #852 - Feature request: Locality names
- Issue #851 - hpx::cout output does not appear on screen
- Issue #849 - All tests fail on OS X after installing
- Issue #848 - Update OS X build instructions
- Issue #846 - Update hpx_external_example
- Issue #845 - Issues with having both debug and release modules in the same directory
- Issue #844 - Create configuration header
- Issue #843 - Tests should use CTest
- Issue #842 - Remove buffer_pool from MPI parcelport
- Issue #841 - Add possibility to broadcast an index with hpx::lcos::broadcast
- Issue #838 - Simplify util::tuple
- Issue #837 - Adopt boost::tuple tests for util::tuple
- Issue #836 - Adopt boost::function tests for util::function
- Issue #835 - Tuple interface missing pieces
- Issue #833 - Partially preprocessing files not working
- Issue #832 - Native papi counters do not work with wild cards
- Issue #831 - Arithmetics counter fails if only one parameter is given
- Issue #830 - Convert hpx::util::function to use new scheme for serializing its base pointer
- Issue #829 - Consistently use decay<T> instead of remove_const< remove_reference<T>>
- Issue #828 - Update future implementation to N3721 and N3722
- Issue #827 - Enable MPI parcelport for bootstrapping whenever application was started using mpirun
- Issue #826 - Support command line option --hpx::print-bind even if --hpx::bind was not used
- Issue #825 - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- Issue #824 - Enable lambda functions to be used with hpx::async/hpx::apply
- Issue #823 - Using a hashing filter
- Issue #822 - Silence unused variable warning
- Issue #821 - Detect if a function object is callable with given arguments
- Issue #820 - Allow wildcards to be used for performance counter names
- Issue #819 - Make the AGAS symbolic name registry distributed
- Issue #818 - Add future::then() overload taking an executor
- Issue #817 - Fixed typo
- Issue #815 - Create an lco that is performing an efficient broadcast of actions
- Issue #814 - Papi counters cannot specify thread#* to get the counts for all threads
- Issue #813 - Scoped unlock
- Issue #811 - simple_central_tuplespace_client run error
- Issue #810 - ostream error when << any objects
- Issue #809 - Optimize parcel serialization
- Issue #808 - HPX applications throw exception when executed from the build directory
- Issue #807 - Create performance counters exposing overall AGAS statistics

4456 https://github.com/STEllAR-GROUP/hpx/issues/831
4457 https://github.com/STEllAR-GROUP/hpx/issues/830
4458 https://github.com/STEllAR-GROUP/hpx/issues/829
4459 https://github.com/STEllAR-GROUP/hpx/issues/828
4460 https://github.com/STEllAR-GROUP/hpx/issues/827
4461 https://github.com/STEllAR-GROUP/hpx/issues/826
4462 https://github.com/STEllAR-GROUP/hpx/issues/825
4463 https://github.com/STEllAR-GROUP/hpx/issues/824
4464 https://github.com/STEllAR-GROUP/hpx/issues/823
4465 https://github.com/STEllAR-GROUP/hpx/issues/822
4466 https://github.com/STEllAR-GROUP/hpx/issues/821
4467 https://github.com/STEllAR-GROUP/hpx/issues/820
4468 https://github.com/STEllAR-GROUP/hpx/issues/819
4469 https://github.com/STEllAR-GROUP/hpx/issues/818
4470 https://github.com/STEllAR-GROUP/hpx/issues/817
4471 https://github.com/STEllAR-GROUP/hpx/issues/815
4472 https://github.com/STEllAR-GROUP/hpx/issues/814
4473 https://github.com/STEllAR-GROUP/hpx/issues/813
4474 https://github.com/STEllAR-GROUP/hpx/issues/811
4475 https://github.com/STEllAR-GROUP/hpx/issues/810
4476 https://github.com/STEllAR-GROUP/hpx/issues/809
4477 https://github.com/STEllAR-GROUP/hpx/issues/808
4478 https://github.com/STEllAR-GROUP/hpx/issues/807
• Issue #795 - Create timed make_ready_future
• Issue #794 - Create heterogeneous when_all/when_any/etc.
• Issue #721 - Make HPX usable for Xeon Phi
• Issue #694 - CMakem should complain if you attempt to build an example without its dependencies
• Issue #692 - SLURM support broken
• Issue #683 - python/hpx/process.py imports epoll on all platforms
• Issue #619 - Automate the doc building process
• Issue #600 - GTC performance broken
• Issue #577 - Allow for zero copy serialization/networking
• Issue #551 - Change executable names to have debug postfix in Debug builds
• Issue #544 - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init
• Issue #534 - hpx::init should take functions by std::function and should accept all forms of hpx_main
• Issue #508 - FindPackage fails to set FOO_LIBRARY_DIR
• Issue #506 - Add cmake support to generate ini files for external applications
• Issue #470 - Changing build-type after configure does not update boost library names
• Issue #453 - Document hpx_run_tests.py
• Issue #445 - Significant performance mismatch between MPI and HPX in SMP for allgather example
• Issue #443 - Make docs viewable from build directory
• Issue #421 - Support multiple HPX instances per node in a batch environment like PBS or SLURM
• Issue #316 - Add message size limitation
• Issue #249 - Clean up locking code in big boot barrier
• Issue #136 - Persistent CMake variables need to be marked as cache variables

https://github.com/STEllAR-GROUP/hpx/issues/795
https://github.com/STEllAR-GROUP/hpx/issues/794
https://github.com/STEllAR-GROUP/hpx/issues/721
https://github.com/STEllAR-GROUP/hpx/issues/694
https://github.com/STEllAR-GROUP/hpx/issues/692
https://github.com/STEllAR-GROUP/hpx/issues/683
https://github.com/STEllAR-GROUP/hpx/issues/619
https://github.com/STEllAR-GROUP/hpx/issues/600
https://github.com/STEllAR-GROUP/hpx/issues/577
https://github.com/STEllAR-GROUP/hpx/issues/551
https://github.com/STEllAR-GROUP/hpx/issues/544
https://github.com/STEllAR-GROUP/hpx/issues/534
https://github.com/STEllAR-GROUP/hpx/issues/508
https://github.com/STEllAR-GROUP/hpx/issues/506
https://github.com/STEllAR-GROUP/hpx/issues/470
https://github.com/STEllAR-GROUP/hpx/issues/453
https://github.com/STEllAR-GROUP/hpx/issues/445
https://github.com/STEllAR-GROUP/hpx/issues/443
https://github.com/STEllAR-GROUP/hpx/issues/421
https://github.com/STEllAR-GROUP/hpx/issues/316
https://github.com/STEllAR-GROUP/hpx/issues/249
https://github.com/STEllAR-GROUP/hpx/issues/136
2.10.19  *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\(^{4501}\) and related proposals to the C++ standardization committee (such as \(N3632^{4502}\) and \(N3857^{4503}\)).

- 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).

- 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 \(N3562^{4504}\). 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\(^{4505}\)).

\(^{4501}\) [http://www.open-std.org/jtc1/sc22/wg21](http://www.open-std.org/jtc1/sc22/wg21)

\(^{4502}\) [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3632.html](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3632.html)

\(^{4503}\) [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf)

\(^{4504}\) [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf)

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 - make (all) in examples folder does nothing
- Issue #805 - Adding the introduction and fixing DOCBOOK dependencies for Windows use
- Issue #804 - Add stackless (non-suspendable) thread type
- Issue #803 - Create proper serialization support functions for util::tuple
- Issue #800 - Add possibility to disable array optimizations during serialization
- Issue #798 - HPX_LIMIT does not work for local dataflow
- Issue #797 - Create a parcelport which uses MPI
- Issue #796 - Problem with Large Numbers of Threads
- Issue #793 - Changing dataflow test case to hang consistently
- Issue #792 - CMake Error
- Issue #791 - Problems with local::dataflow
- Issue #790 - wait_for() doesn’t compile
- Issue #789 - HPX with Intel compiler segfaults
- Issue #788 - Intel compiler support
- Issue #787 - Fixed SFINAEd specializations
- Issue #786 - Memory issues during benchmarking.
- Issue #785 - Create an API allowing to register external threads with HPX
- Issue #784 - util::plugin is throwing an error when a symbol is not found
- Issue #783 - How does hpx::bind work?
- Issue #782 - Added quotes around STRING REPLACE potentially empty arguments
- Issue #781 - Make sure no exceptions propagate into the thread manager

https://github.com/STEllAR-GROUP/hpx/issues/806
https://github.com/STEllAR-GROUP/hpx/issues/805
https://github.com/STEllAR-GROUP/hpx/issues/804
https://github.com/STEllAR-GROUP/hpx/issues/803
https://github.com/STEllAR-GROUP/hpx/issues/798
https://github.com/STEllAR-GROUP/hpx/issues/797
https://github.com/STEllAR-GROUP/hpx/issues/796
https://github.com/STEllAR-GROUP/hpx/issues/793
https://github.com/STEllAR-GROUP/hpx/issues/792
https://github.com/STEllAR-GROUP/hpx/issues/791
https://github.com/STEllAR-GROUP/hpx/issues/790
https://github.com/STEllAR-GROUP/hpx/issues/789
https://github.com/STEllAR-GROUP/hpx/issues/788
https://github.com/STEllAR-GROUP/hpx/issues/787
https://github.com/STEllAR-GROUP/hpx/issues/786
https://github.com/STEllAR-GROUP/hpx/issues/785
https://github.com/STEllAR-GROUP/hpx/issues/784
https://github.com/STEllAR-GROUP/hpx/issues/783
https://github.com/STEllAR-GROUP/hpx/issues/782
https://github.com/STEllAR-GROUP/hpx/issues/781
• Issue #780[^527] - Allow arithmetics performance counters to expand its parameters
• Issue #779[^528] - Test case for 778
• Issue #778[^529] - Swapping futures segfaults
• Issue #777[^530] - hpx::lcos::details::when_xxx don’t restore completion handlers
• Issue #776[^531] - Compiler chokes on dataflow overload with launch policy
• Issue #775[^532] - Runtime error with local dataflow (copying futures?)
• Issue #774[^533] - Using local dataflow without explicit namespace
• Issue #773[^534] - Local dataflow with unwrap: functor operators need to be const
• Issue #772[^535] - Allow (remote) actions to return a future
• Issue #771[^536] - Setting HPX_LIMIT gives huge boost MPL errors
• Issue #770[^537] - Add launch policy to (local) dataflow
• Issue #769[^538] - Make compile time configuration information available
• Issue #768[^539] - Const correctness problem in local dataflow
• Issue #767[^540] - Add launch policies to async
• Issue #766[^541] - Mark data structures for optimized (array based) serialization
• Issue #765[^542] - Align hpx::any with N3508: Any Library Proposal (Revision 2)
• Issue #764[^543] - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations
• Issue #762[^544] - added a human readable output for the ping pong example
• Issue #761[^545] - Ambiguous typename when constructing derived component
• Issue #760[^546] - Simple components can not be derived
• Issue #759[^547] - make install doesn’t give a complete install
• Issue #758[^548] - Stack overflow when using locking_hook<>
• Issue #757[^549] - copy paste error; unsupported function overloading

[^527]: https://github.com/STEllAR-GROUP/hpx/issues/780
[^528]: https://github.com/STEllAR-GROUP/hpx/issues/779
[^529]: https://github.com/STEllAR-GROUP/hpx/issues/778
[^530]: https://github.com/STEllAR-GROUP/hpx/issues/777
[^531]: https://github.com/STEllAR-GROUP/hpx/issues/776
[^532]: https://github.com/STEllAR-GROUP/hpx/issues/775
[^533]: https://github.com/STEllAR-GROUP/hpx/issues/774
[^534]: https://github.com/STEllAR-GROUP/hpx/issues/773
[^535]: https://github.com/STEllAR-GROUP/hpx/issues/772
[^536]: https://github.com/STEllAR-GROUP/hpx/issues/771
[^537]: https://github.com/STEllAR-GROUP/hpx/issues/770
[^538]: https://github.com/STEllAR-GROUP/hpx/issues/769
[^539]: https://github.com/STEllAR-GROUP/hpx/issues/768
[^540]: https://github.com/STEllAR-GROUP/hpx/issues/767
[^541]: https://github.com/STEllAR-GROUP/hpx/issues/766
[^542]: https://github.com/STEllAR-GROUP/hpx/issues/765
[^543]: https://github.com/STEllAR-GROUP/hpx/issues/764
[^544]: https://github.com/STEllAR-GROUP/hpx/issues/762
[^545]: https://github.com/STEllAR-GROUP/hpx/issues/761
[^546]: https://github.com/STEllAR-GROUP/hpx/issues/760
[^547]: https://github.com/STEllAR-GROUP/hpx/issues/759
[^548]: https://github.com/STEllAR-GROUP/hpx/issues/758
[^549]: https://github.com/STEllAR-GROUP/hpx/issues/757
• Issue #756 - GTCX runtime issue in Gordon
• Issue #755 - Papi counters don’t work with reset and evaluate API’s
• Issue #753 - cmake bugfix and improved component action docs
• Issue #752 - hpx simple component docs
• Issue #750 - Add hpx::util::any
• Issue #749 - Thread phase counter is not reset
• Issue #748 - Memory performance counter are not registered
• Issue #747 - Create performance counters exposing arithmetic operations
• Issue #745 - apply_callback needs to invoke callback when applied locally
• Issue #744 - CMake fixes
• Issue #743 - Problem Building github version of HPX
• Issue #742 - Remove HPX_STD_BIND
• Issue #741 - assertion ‘px != 0’ failed: HPX(assertion_failure) for low numbers of OS threads
• Issue #739 - Performance counters do not count to the end of the program or evaluation
• Issue #738 - Dedicated AGAS server runs don’t work; console ignores -a option.
• Issue #737 - Missing bind overloads
• Issue #736 - Performance counter wildcards do not always work
• Issue #735 - Create native ibverbs parcelport based on rdma operations
• Issue #734 - Threads stolen performance counter total is incorrect
• Issue #733 - Test benchmarks need to be checked and fixed
• Issue #732 - Build fails with Mac, using mac ports clang-3.3 on latest git branch
• Issue #731 - Add global start/stop API for performance counters
• Issue #730 - Performance counter values are apparently incorrect

4550 https://github.com/STEllAR-GROUP/hpx/issues/756
4551 https://github.com/STEllAR-GROUP/hpx/issues/755
4552 https://github.com/STEllAR-GROUP/hpx/issues/753
4553 https://github.com/STEllAR-GROUP/hpx/issues/752
4554 https://github.com/STEllAR-GROUP/hpx/issues/750
4555 https://github.com/STEllAR-GROUP/hpx/issues/749
4556 https://github.com/STEllAR-GROUP/hpx/issues/748
4557 https://github.com/STEllAR-GROUP/hpx/issues/747
4558 https://github.com/STEllAR-GROUP/hpx/issues/745
4559 https://github.com/STEllAR-GROUP/hpx/issues/744
4560 https://github.com/STEllAR-GROUP/hpx/issues/743
4561 https://github.com/STEllAR-GROUP/hpx/issues/742
4562 https://github.com/STEllAR-GROUP/hpx/issues/741
4563 https://github.com/STEllAR-GROUP/hpx/issues/739
4564 https://github.com/STEllAR-GROUP/hpx/issues/738
4565 https://github.com/STEllAR-GROUP/hpx/issues/737
4566 https://github.com/STEllAR-GROUP/hpx/issues/736
4567 https://github.com/STEllAR-GROUP/hpx/issues/735
4568 https://github.com/STEllAR-GROUP/hpx/issues/734
4569 https://github.com/STEllAR-GROUP/hpx/issues/733
4570 https://github.com/STEllAR-GROUP/hpx/issues/732
4571 https://github.com/STEllAR-GROUP/hpx/issues/731
4572 https://github.com/STEllAR-GROUP/hpx/issues/730
• Issue #729 - Unhandled switch
• Issue #728 - Serialization of hpx::util::function between two localities causes seg faults
• Issue #727 - Memory counters on Mac OS X
• Issue #725 - Restore original thread priority on resume
• Issue #724 - Performance benchmarks do not depend on main HPX libraries
• Issue #723 - [teletype]\-hpx:nodes=``cat $PBS_NODEFILE`` works; –hpx:nodefile=$PBS_NODEFILE does not.[c++]
• Issue #722 - Fix binding const member functions as actions
• Issue #719 - Create performance counter exposing compression ratio
• Issue #718 - Add possibility to compress parcel data
• Issue #717 - strip_credit_from_gid has misleading semantics
• Issue #716 - Non-option arguments to programs run using pbsdsh must be before --hpx:nodes, contrary to directions
• Issue #715 - Re-thrown exceptions should retain the original call site
• Issue #714 - failed assertion in debug mode
• Issue #713 - Add performance counters monitoring connection caches
• Issue #712 - Adjust parcel related performance counters to be connection type specific
• Issue #711 - configuration failure
• Issue #710 - Error “timed out while trying to find room in the connection cache” when trying to start multiple localities on a single computer
• Issue #709 - Add new thread state ‘staged’ referring to task descriptions
• Issue #708 - Detect/mitigate bad non-system installs of GCC on Redhat systems
• Issue #707 - Many examples do not link with Git HEAD version
• Issue #706 - hpx::init removes portions of non-option command line arguments before last = sign
• Issue #705 - Create rolling average and median aggregating performance counters

---

4573 https://github.com/STEllAR-GROUP/hpx/issues/729
4574 https://github.com/STEllAR-GROUP/hpx/issues/728
4575 https://github.com/STEllAR-GROUP/hpx/issues/727
4576 https://github.com/STEllAR-GROUP/hpx/issues/725
4577 https://github.com/STEllAR-GROUP/hpx/issues/724
4578 https://github.com/STEllAR-GROUP/hpx/issues/723
4579 https://github.com/STEllAR-GROUP/hpx/issues/722
4580 https://github.com/STEllAR-GROUP/hpx/issues/719
4581 https://github.com/STEllAR-GROUP/hpx/issues/718
4582 https://github.com/STEllAR-GROUP/hpx/issues/717
4583 https://github.com/STEllAR-GROUP/hpx/issues/716
4584 https://github.com/STEllAR-GROUP/hpx/issues/715
4585 https://github.com/STEllAR-GROUP/hpx/issues/714
4586 https://github.com/STEllAR-GROUP/hpx/issues/713
4587 https://github.com/STEllAR-GROUP/hpx/issues/712
4588 https://github.com/STEllAR-GROUP/hpx/issues/711
4589 https://github.com/STEllAR-GROUP/hpx/issues/710
4590 https://github.com/STEllAR-GROUP/hpx/issues/709
4591 https://github.com/STEllAR-GROUP/hpx/issues/708
4592 https://github.com/STEllAR-GROUP/hpx/issues/707
4593 https://github.com/STEllAR-GROUP/hpx/issues/706
4594 https://github.com/STEllAR-GROUP/hpx/issues/705

---

1806 Chapter 2. What’s so special about HPX?
• Issue #704 - Create performance counter to expose thread queue waiting time
• Issue #703 - Add support to HPX build system to find librctool.a and related headers
• Issue #699 - Generalize instrumentation support
• Issue #698 - compilation failure with hwloc absent
• Issue #697 - Performance counter counts should be zero indexed
• Issue #696 - Distributed problem
• Issue #695 - Bad perf counter time printed
• Issue #693 - --help doesn’t print component specific command line options
• Issue #692 - SLURM support broken
• Issue #691 - exception while executing any application linked with hwloc
• Issue #690 - thread_id_test and thread_launcher_test failing
• Issue #689 - Make the buildbots use hwloc
• Issue #687 - compilation error fix (hwloc_topology)
• Issue #686 - Linker Error for Applications
• Issue #684 - Pinning of service thread fails when number of worker threads equals the number of cores
• Issue #682 - Add performance counters exposing number of stolen threads
• Issue #681 - Add apply_continue for asynchronous chaining of actions
• Issue #679 - Remove obsolete async_callback API functions
• Issue #678 - Add new API for setting/triggering LCOs
• Issue #677 - Add async_continue for true continuation style actions
• Issue #676 - Buildbot for gcc 4.4 broken
• Issue #675 - Partial preprocessing broken
• Issue #674 - HPX segfaults when built with gcc 4.7
HPX Documentation, master

- Issue #673\textsuperscript{4618} - use\_guard\_pages has inconsistent preprocessor guards
- Issue #672\textsuperscript{4619} - External build breaks if library path has spaces
- Issue #671\textsuperscript{4620} - release tarballs are tarbombs
- Issue #670\textsuperscript{4621} - CMake won’t find Boost headers in layout=versioned install
- Issue #669\textsuperscript{4622} - Links in docs to source files broken if not installed
- Issue #667\textsuperscript{4623} - Not reading ini file properly
- Issue #664\textsuperscript{4624} - Adapt new meanings of ‘const’ and ‘mutable’
- Issue #661\textsuperscript{4625} - Implement BTL Parcel port
- Issue #655\textsuperscript{4626} - Make HPX work with the “decltype” result\_of
- Issue #647\textsuperscript{4627} - documentation for specifying the number of high priority threads
  --hpx:high-priority-threads
- Issue #643\textsuperscript{4628} - Error parsing host file
- Issue #642\textsuperscript{4629} - HWLoc issue with TAU
- Issue #639\textsuperscript{4630} - Logging potentially suspends a running thread
- Issue #634\textsuperscript{4631} - Improve error reporting from parcel layer
- Issue #627\textsuperscript{4632} - Add tests for async and apply overloads that accept regular C++ functions
- Issue #626\textsuperscript{4633} - hpx/future.hpp header
- Issue #601\textsuperscript{4634} - Intel support
- Issue #557\textsuperscript{4635} - Remove action codes
- Issue #531\textsuperscript{4636} - AGAS request and response classes should use switch statements
- Issue #529\textsuperscript{4637} - Investigate the state of hwloc support
- Issue #526\textsuperscript{4638} - Make HPX aware of hyper-threading
- Issue #518\textsuperscript{4639} - Create facilities allowing to use plain arrays as action arguments
- Issue #473\textsuperscript{4640} - hwloc thread binding is broken on CPUs with hyperthreading

\textsuperscript{4618} https://github.com/STEllAR-GROUP/hpx/issues/673
\textsuperscript{4619} https://github.com/STEllAR-GROUP/hpx/issues/672
\textsuperscript{4620} https://github.com/STEllAR-GROUP/hpx/issues/671
\textsuperscript{4621} https://github.com/STEllAR-GROUP/hpx/issues/670
\textsuperscript{4622} https://github.com/STEllAR-GROUP/hpx/issues/669
\textsuperscript{4623} https://github.com/STEllAR-GROUP/hpx/issues/667
\textsuperscript{4624} https://github.com/STEllAR-GROUP/hpx/issues/664
\textsuperscript{4625} https://github.com/STEllAR-GROUP/hpx/issues/661
\textsuperscript{4626} https://github.com/STEllAR-GROUP/hpx/issues/655
\textsuperscript{4627} https://github.com/STEllAR-GROUP/hpx/issues/647
\textsuperscript{4628} https://github.com/STEllAR-GROUP/hpx/issues/643
\textsuperscript{4629} https://github.com/STEllAR-GROUP/hpx/issues/642
\textsuperscript{4630} https://github.com/STEllAR-GROUP/hpx/issues/639
\textsuperscript{4631} https://github.com/STEllAR-GROUP/hpx/issues/634
\textsuperscript{4632} https://github.com/STEllAR-GROUP/hpx/issues/627
\textsuperscript{4633} https://github.com/STEllAR-GROUP/hpx/issues/626
\textsuperscript{4634} https://github.com/STEllAR-GROUP/hpx/issues/601
\textsuperscript{4635} https://github.com/STEllAR-GROUP/hpx/issues/557
\textsuperscript{4636} https://github.com/STEllAR-GROUP/hpx/issues/531
\textsuperscript{4637} https://github.com/STEllAR-GROUP/hpx/issues/529
\textsuperscript{4638} https://github.com/STEllAR-GROUP/hpx/issues/526
\textsuperscript{4639} https://github.com/STEllAR-GROUP/hpx/issues/518
\textsuperscript{4640} https://github.com/STEllAR-GROUP/hpx/issues/473

1808 Chapter 2. What’s so special about HPX?
2.10.20  **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 improvements. 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.

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. 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).
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  - Segfault on calling hpx::finalize twice
- Issue #665  - Adding declaration num_of_cores
- Issue #662  - pkgconfig is building wrong
- Issue #660  - Need uninterrupt function
- Issue #659  - Move our logging library into a different namespace
- Issue #658  - Dynamic performance counter types are broken
- Issue #657  - HPX v0.9.5 (RC1) hello_world example segfaulting
- Issue #656  - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- Issue #654  - Integrate the Boost auto_index tool with documentation
- Issue #653  - Make HPX build on OS X + Clang + libc++
- Issue #651  - Add fine-grained control for thread pinning
- Issue #650  - Command line no error message when using -hpx:(anything)
- Issue #645  - Command line aliases don’t work in [teletype]\`@file\`[c++]
- Issue #644  - Terminated threads are not always properly cleaned up
- Issue #640  - future_data<T>::set_on_completed_ used without locks
- Issue #638  - hpx build with intel compilers fails on linux
- Issue #637  - --copy-dt-needed-entries breaks with gold
- Issue #635  - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- Issue #633  - Re-add examples to final 0.9.5 release
- Issue #632  - Example thread_aware_timer is broken
- Issue #631  - FFT application throws error in parcellayer

https://github.com/STEllAR-GROUP/hpx/issues/666
https://github.com/STEllAR-GROUP/hpx/issues/665
https://github.com/STEllAR-GROUP/hpx/issues/662
https://github.com/STEllAR-GROUP/hpx/issues/660
https://github.com/STEllAR-GROUP/hpx/issues/659
https://github.com/STEllAR-GROUP/hpx/issues/658
https://github.com/STEllAR-GROUP/hpx/issues/657
https://github.com/STEllAR-GROUP/hpx/issues/656
https://github.com/STEllAR-GROUP/hpx/issues/654
https://github.com/STEllAR-GROUP/hpx/issues/653
https://github.com/STEllAR-GROUP/hpx/issues/651
https://github.com/STEllAR-GROUP/hpx/issues/650
https://github.com/STEllAR-GROUP/hpx/issues/645
https://github.com/STEllAR-GROUP/hpx/issues/644
https://github.com/STEllAR-GROUP/hpx/issues/640
https://github.com/STEllAR-GROUP/hpx/issues/638
https://github.com/STEllAR-GROUP/hpx/issues/637
https://github.com/STEllAR-GROUP/hpx/issues/635
https://github.com/STEllAR-GROUP/hpx/issues/633
https://github.com/STEllAR-GROUP/hpx/issues/632
https://github.com/STEllAR-GROUP/hpx/issues/631
• Issue #630\(^{4669}\) - Event synchronization example is broken
• Issue #629\(^{4670}\) - Waiting on futures hangs
• Issue #628\(^{4671}\) - Add an `HPX_ALWAYS_ASSERT` macro
• Issue #625\(^{4672}\) - Port coroutines context switch benchmark
• Issue #621\(^{4673}\) - New INI section for stack sizes
• Issue #618\(^{4674}\) - pkg_config support does not work with a HPX debug build
• Issue #617\(^{4675}\) - hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: ‘get_thread_id’ was not declared in this scope
• Issue #616\(^{4676}\) - Change wait_xxx not to use locking
• Issue #615\(^{4677}\) - Revert visibility ‘fix’ (fb0b6b8245dad1127b0c25eba9d9386b3945cca9)
• Issue #614\(^{4678}\) - Fix Dataflow linker error
• Issue #613\(^{4679}\) - find_here should throw an exception on failure
• Issue #612\(^{4680}\) - Thread phase doesn’t show up in debug mode
• Issue #611\(^{4681}\) - Make stack guard pages configurable at runtime (initialization time)
• Issue #610\(^{4682}\) - Co-Locate Components
• Issue #609\(^{4683}\) - future_overhead
• Issue #608\(^{4684}\) - --hpx:list-counter-infos problem
• Issue #607\(^{4685}\) - Update Boost.Context based backend for coroutines
• Issue #606\(^{4686}\) - 1d_wave_equation is not working
• Issue #605\(^{4687}\) - Any C++ function that has serializable arguments and a serializable return type should be remotable
• Issue #604\(^{4688}\) - Connecting localities isn’t working anymore
• Issue #603\(^{4689}\) - Do not verify any ini entries read from a file
• Issue #602\(^{4690}\) - Rename argument_size to type_size/ added implementation to get parcel size

\(^{4669}\) https://github.com/STEllAR-GROUP/hpx/issues/630
^{4670} https://github.com/STEllAR-GROUP/hpx/issues/629
^{4671} https://github.com/STEllAR-GROUP/hpx/issues/628
^{4672} https://github.com/STEllAR-GROUP/hpx/issues/625
^{4673} https://github.com/STEllAR-GROUP/hpx/issues/621
^{4674} https://github.com/STEllAR-GROUP/hpx/issues/618
^{4675} https://github.com/STEllAR-GROUP/hpx/issues/617
^{4676} https://github.com/STEllAR-GROUP/hpx/issues/616
^{4677} https://github.com/STEllAR-GROUP/hpx/issues/615
^{4678} https://github.com/STEllAR-GROUP/hpx/issues/614
^{4679} https://github.com/STEllAR-GROUP/hpx/issues/613
^{4680} https://github.com/STEllAR-GROUP/hpx/issues/612
^{4681} https://github.com/STEllAR-GROUP/hpx/issues/611
^{4682} https://github.com/STEllAR-GROUP/hpx/issues/610
^{4683} https://github.com/STEllAR-GROUP/hpx/issues/609
^{4684} https://github.com/STEllAR-GROUP/hpx/issues/608
^{4685} https://github.com/STEllAR-GROUP/hpx/issues/607
^{4686} https://github.com/STEllAR-GROUP/hpx/issues/606
^{4687} https://github.com/STEllAR-GROUP/hpx/issues/605
^{4688} https://github.com/STEllAR-GROUP/hpx/issues/604
^{4689} https://github.com/STEllAR-GROUP/hpx/issues/603
^{4690} https://github.com/STEllAR-GROUP/hpx/issues/602

2.10. Releases
- Issue #599 - Enable locality specific command line options
- Issue #598 - Need an API that accesses the performance counter reporting the system uptime
- Issue #597 - Compiling on ranger
- Issue #595 - I need a place to store data in a thread self pointer
- Issue #594 - 32/64 interoperability
- Issue #593 - Warn if logging is disabled at compile time but requested at runtime
- Issue #592 - Add optional argument value to --hpx:list-counters and --hpx:list-counter-infos
- Issue #591 - Allow for wildcards in performance counter names specified with --hpx:print-counter
- Issue #590 - Local promise semantic differences
- Issue #589 - Create API to query performance counter names
- Issue #587 - Add get_num_localities and get_num_threads to AGAS API
- Issue #586 - Adjust local AGAS cache size based on number of localities
- Issue #585 - Error while using counters in HPX
- Issue #584 - Counting argument size of actions, initial pass.
- Issue #581 - Remove RemoteResult template parameter for future<br>
- Issue #580 - Add possibility to hook into actions
- Issue #578 - Use angle brackets in HPX error dumps
- Issue #576 - Exception incorrectly thrown when --help is used
- Issue #575 - HPX(bad_component_type) with gcc 4.7.2 and boost 1.51
- Issue #574 - --hpx:connect command line parameter not working correctly
- Issue #571 - hp::wait() (callback version) should pass the future to the callback function
- Issue #570 - hp::wait should operate on boost::arrays and std::lists
- Issue #569 - Add a logging sink for Android

https://github.com/STEllAR-GROUP/hpx/issues/599
https://github.com/STEllAR-GROUP/hpx/issues/598
https://github.com/STEllAR-GROUP/hpx/issues/597
https://github.com/STEllAR-GROUP/hpx/issues/595
https://github.com/STEllAR-GROUP/hpx/issues/594
https://github.com/STEllAR-GROUP/hpx/issues/593
https://github.com/STEllAR-GROUP/hpx/issues/592
https://github.com/STEllAR-GROUP/hpx/issues/591
https://github.com/STEllAR-GROUP/hpx/issues/590
https://github.com/STEllAR-GROUP/hpx/issues/589
https://github.com/STEllAR-GROUP/hpx/issues/587
https://github.com/STEllAR-GROUP/hpx/issues/586
https://github.com/STEllAR-GROUP/hpx/issues/585
https://github.com/STEllAR-GROUP/hpx/issues/584
https://github.com/STEllAR-GROUP/hpx/issues/581
https://github.com/STEllAR-GROUP/hpx/issues/580
https://github.com/STEllAR-GROUP/hpx/issues/578
https://github.com/STEllAR-GROUP/hpx/issues/576
https://github.com/STEllAR-GROUP/hpx/issues/575
https://github.com/STEllAR-GROUP/hpx/issues/574
https://github.com/STEllAR-GROUP/hpx/issues/571
https://github.com/STEllAR-GROUP/hpx/issues/570
https://github.com/STEllAR-GROUP/hpx/issues/569
• Issue #568[^14] - 2-argument version of `HPX_DEFINE_COMPONENT_ACTION`
• Issue #567[^15] - Connecting to a running HPX application works only once
• Issue #565[^16] - HPX doesn’t shutdown properly
• Issue #564[^17] - Partial preprocessing of new component creation interface
• Issue #563[^18] - Add `hpx::start/hpx::stop` to avoid blocking main thread
• Issue #562[^19] - All command line arguments swallowed by `hpx`
• Issue #561[^20] - Boost.Tuple is not move aware
• Issue #558[^21] - `boost::shared_ptr<>` style semantics/syntax for client classes
• Issue #556[^22] - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
• Issue #555[^23] - `BOOST_FORCEINLINE` does not name a type
• Issue #554[^24] - Possible race condition in thread `get_id()`
• Issue #552[^25] - Move enable `client_base`
• Issue #550[^26] - Add stack size category ‘huge’
• Issue #549[^27] - ShenEOS run seg-faults on single or distributed runs
• Issue #545[^28] - `AUTOGLOB` broken for `add_hpx_component`
• Issue #542[^29] - `FindHPX_HDF5` still searches multiple times
• Issue #541[^30] - Quotes around application name in `hpx::init`
• Issue #539[^31] - Race condition occurring with new lightweight threads
• Issue #535[^32] - `hpx_run_tests.py` exits with no error code when tests are missing
• Issue #530[^33] - Thread description(unknown) in logs
• Issue #523[^34] - Make thread objects more lightweight
• Issue #521[^35] - `hpx::error_code` is not usable for lightweight error handling
• Issue #520[^36] - Add full user environment to HPX logs

[^14]: https://github.com/STEllAR-GROUP/hpx/issues/568
[^15]: https://github.com/STEllAR-GROUP/hpx/issues/567
[^16]: https://github.com/STEllAR-GROUP/hpx/issues/565
[^17]: https://github.com/STEllAR-GROUP/hpx/issues/564
[^18]: https://github.com/STEllAR-GROUP/hpx/issues/563
[^19]: https://github.com/STEllAR-GROUP/hpx/issues/562
[^20]: https://github.com/STEllAR-GROUP/hpx/issues/561
[^21]: https://github.com/STEllAR-GROUP/hpx/issues/558
[^22]: https://github.com/STEllAR-GROUP/hpx/issues/556
[^23]: https://github.com/STEllAR-GROUP/hpx/issues/555
[^24]: https://github.com/STEllAR-GROUP/hpx/issues/554
[^25]: https://github.com/STEllAR-GROUP/hpx/issues/552
[^26]: https://github.com/STEllAR-GROUP/hpx/issues/550
[^27]: https://github.com/STEllAR-GROUP/hpx/issues/549
[^28]: https://github.com/STEllAR-GROUP/hpx/issues/545
[^29]: https://github.com/STEllAR-GROUP/hpx/issues/542
[^30]: https://github.com/STEllAR-GROUP/hpx/issues/541
[^31]: https://github.com/STEllAR-GROUP/hpx/issues/539
[^32]: https://github.com/STEllAR-GROUP/hpx/issues/535
[^33]: https://github.com/STEllAR-GROUP/hpx/issues/530
[^34]: https://github.com/STEllAR-GROUP/hpx/issues/523
[^35]: https://github.com/STEllAR-GROUP/hpx/issues/521
[^36]: https://github.com/STEllAR-GROUP/hpx/issues/520
• Issue #519\textsuperscript{4737} - Build succeeds, running fails
• Issue #517\textsuperscript{4738} - Add a guard page to linux coroutine stacks
• Issue #516\textsuperscript{4739} - hpx::thread::detach suspends while holding locks, leads to hang in debug
• Issue #514\textsuperscript{4740} - Preprocessed headers for <hpx/apply.hpp> don’t compile
• Issue #513\textsuperscript{4741} - Buildbot configuration problem
• Issue #512\textsuperscript{4742} - Implement action based stack size customization
• Issue #511\textsuperscript{4743} - Move action priority into a separate type trait
• Issue #510\textsuperscript{4744} - trunk broken
• Issue #507\textsuperscript{4745} - no matching function for call to boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(hpx::threads::linux_topology*)
• Issue #505\textsuperscript{4746} - undefined_symbol regression test currently failing
• Issue #502\textsuperscript{4747} - Adding OpenCL and OCLM support to HPX for Windows and Linux
• Issue #501\textsuperscript{4748} - find_package(HPX) sets cmake output variables
• Issue #500\textsuperscript{4749} - wait_any/wait_all are badly named
• Issue #499\textsuperscript{4750} - Add support for disabling pbs support in pbs runs
• Issue #498\textsuperscript{4751} - Error during no-cache runs
• Issue #496\textsuperscript{4752} - Add partial preprocessing support to cmake
• Issue #495\textsuperscript{4753} - Support HPX modules exporting startup/shutdown functions only
• Issue #494\textsuperscript{4754} - Allow modules to specify when to run startup/shutdown functions
• Issue #493\textsuperscript{4755} - Avoid constructing a string in make_success_code
• Issue #492\textsuperscript{4756} - Performance counter creation is no longer synchronized at startup
• Issue #491\textsuperscript{4757} - Performance counter creation is no longer synchronized at startup
• Issue #490\textsuperscript{4758} - Sheneos on_completed_bulk seg fault in distributed
• Issue #489\textsuperscript{4759} - compiling issue with g++44
• Issue #488 - Adding OpenCL and OCLM support to HPX for the MSVC platform
• Issue #487 - FindHPX.cmake problems
• Issue #485 - Change distributing_factor and binpacking_factor to use bulk creation
• Issue #484 - Change HPX\_DONT\_USE\_PREPROCESSED\_FILES to HPX\_USE\_PREPROCESSED\_FILES
• Issue #483 - Memory counter for Windows
• Issue #479 - Strange errors appear when requesting performance counters on multiple nodes
• Issue #477 - Create (global) timer for multi-threaded measurements
• Issue #472 - Add partial preprocessing using Wave
• Issue #471 - Segfault stack traces don’t show up in release
• Issue #468 - External projects need to link with internal components
• Issue #462 - Startup/shutdown functions are called more than once
• Issue #458 - Consolidate hpx::util::high_resolution_timer and hpx::util::high_resolution_clock
• Issue #457 - Index out of bounds in allgather\_and\_gate on 4 cores or more
• Issue #448 - Make HPX compile with clang
• Issue #447 - ‘make tests’ should execute tests on local installation
• Issue #446 - Remove SVN-related code from the codebase
• Issue #444 - Race condition in smp
• Issue #441 - Patched Boost.Serialization headers should only be installed if needed
• Issue #439 - Components using HPX\_REGISTER\_STARTUP\_MODULE fail to compile with MSVC
• Issue #436 - Verify that no locks are being held while threads are suspended
• Issue #435 - Installing HPX should not clobber existing Boost installation
• Issue #434 - Logging external component failed (Boost 1.50)
• Issue #433 - Runtime crash when building all examples

4760 https://github.com/STEllAR-GROUP/hpx/issues/488
4761 https://github.com/STEllAR-GROUP/hpx/issues/487
4762 https://github.com/STEllAR-GROUP/hpx/issues/485
4763 https://github.com/STEllAR-GROUP/hpx/issues/484
4764 https://github.com/STEllAR-GROUP/hpx/issues/483
4765 https://github.com/STEllAR-GROUP/hpx/issues/479
4766 https://github.com/STEllAR-GROUP/hpx/issues/477
4767 https://github.com/STEllAR-GROUP/hpx/issues/472
4768 https://github.com/STEllAR-GROUP/hpx/issues/471
4769 https://github.com/STEllAR-GROUP/hpx/issues/468
4770 https://github.com/STEllAR-GROUP/hpx/issues/462
4771 https://github.com/STEllAR-GROUP/hpx/issues/458
4772 https://github.com/STEllAR-GROUP/hpx/issues/457
4773 https://github.com/STEllAR-GROUP/hpx/issues/448
4774 https://github.com/STEllAR-GROUP/hpx/issues/447
4775 https://github.com/STEllAR-GROUP/hpx/issues/446
4776 https://github.com/STEllAR-GROUP/hpx/issues/444
4777 https://github.com/STEllAR-GROUP/hpx/issues/441
4778 https://github.com/STEllAR-GROUP/hpx/issues/439
4779 https://github.com/STEllAR-GROUP/hpx/issues/436
4780 https://github.com/STEllAR-GROUP/hpx/issues/435
4781 https://github.com/STEllAR-GROUP/hpx/issues/434
4782 https://github.com/STEllAR-GROUP/hpx/issues/433
• Issue #432[^4783] - Dataflow hangs on 512 cores/64 nodes
• Issue #430[^4784] - Problem with distributing factory
• Issue #424[^4785] - File paths referring to XSL-files need to be properly escaped
• Issue #417[^4786] - Make dataflow LCOs work out of the box by using partial preprocessing
• Issue #413[^4787] - hpx_svnversion.py fails on Windows
• Issue #412[^4788] - Make hpx::error_code equivalent to hpx::exception
• Issue #398[^4789] - HPX clobbers out-of-tree application specific CMake variables (specifically CMAKE_BUILD_TYPE)
• Issue #394[^4790] - Remove code generating random port numbers for network
• Issue #378[^4791] - ShenEOS scaling issues
• Issue #354[^4792] - Create a coroutines wrapper for Boost.Context
• Issue #349[^4793] - Commandline option --localities=N/-1N should be necessary only on AGAS locality
• Issue #334[^4794] - Add auto_index support to cmake based documentation toolchain
• Issue #318[^4795] - Network benchmarks
• Issue #317[^4796] - Implement network performance counters
• Issue #310[^4797] - Duplicate logging entries
• Issue #230[^4798] - Add compile time option to disable thread debugging info
• Issue #171[^4799] - Add an INI option to turn off deadlock detection independently of logging
• Issue #170[^4800] - OSHL internal counters are incorrect
• Issue #103[^4801] - Better diagnostics for multiple component/action registrations under the same name
• Issue #48[^4802] - Support for Darwin (Xcode + Clang)
• Issue #21[^4803] - Build fails with GCC 4.6

[^4783]: https://github.com/STEllAR-GROUP/hpx/issues/432
[^4784]: https://github.com/STEllAR-GROUP/hpx/issues/430
[^4785]: https://github.com/STEllAR-GROUP/hpx/issues/424
[^4786]: https://github.com/STEllAR-GROUP/hpx/issues/417
[^4787]: https://github.com/STEllAR-GROUP/hpx/issues/413
[^4788]: https://github.com/STEllAR-GROUP/hpx/issues/412
[^4789]: https://github.com/STEllAR-GROUP/hpx/issues/398
[^4790]: https://github.com/STEllAR-GROUP/hpx/issues/394
[^4791]: https://github.com/STEllAR-GROUP/hpx/issues/378
[^4792]: https://github.com/STEllAR-GROUP/hpx/issues/354
[^4793]: https://github.com/STEllAR-GROUP/hpx/issues/349
[^4794]: https://github.com/STEllAR-GROUP/hpx/issues/318
[^4795]: https://github.com/STEllAR-GROUP/hpx/issues/317
[^4796]: https://github.com/STEllAR-GROUP/hpx/issues/310
[^4797]: https://github.com/STEllAR-GROUP/hpx/issues/230
[^4798]: https://github.com/STEllAR-GROUP/hpx/issues/171
[^4800]: https://github.com/STEllAR-GROUP/hpx/issues/48
[^4803]: https://github.com/STEllAR-GROUP/hpx/issues/21
2.10.21 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\(^{4804}\).
- 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:

```cpp
#include <hpx/hpx_init.hpp>

int hpx_main()
{
    return hpx::finalize();
}
```

\(^{4804}\) http://www.open-std.org/jtc1/sc22/wg21
This removes the immediate dependency on the **Boost.Program Options** 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**, the minimal program can be written as:

```cpp
#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 the number of messages sent and received, the number of parcels 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`. 

---

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release:

- **Issue #71**[^71] - GIDs that are not serialized via handle_gid<> should raise an exception
- **Issue #105**[^105] - Allow for hpx::util::functions to be registered in the AGAS symbolic namespace
- **Issue #107**[^107] - Nasty threadmanger race condition (reproducible in sheneos_test)
- **Issue #108**[^108] - Add millisecond resolution to HPX logs on Linux
- **Issue #110**[^110] - Shutdown hang in distributed with release build
- **Issue #116**[^116] - Don’t use TSS for the applier and runtime pointers
- **Issue #162**[^162] - Move local synchronous execution shortcut from hpx::function to the applier
- **Issue #172**[^172] - Cache sources in CMake and check if they change manually
- **Issue #178**[^178] - Add an INI option to turn off ranged-based AGAS caching
- **Issue #187**[^187] - Support for disabling performance counter deployment
- **Issue #202**[^202] - Support for sending performance counter data to a specific file
- **Issue #218**[^218] - boost.coroutines allows different stack sizes, but stack pool is unaware of this
- **Issue #231**[^231] - Implement movable boost::bind
- **Issue #232**[^232] - Implement movable boost::function
- **Issue #236**[^236] - Allow binding hpx::util::function to actions
- **Issue #239**[^239] - Replace hpx::function with hpx::util::function
- **Issue #240**[^240] - Can’t specify RemoteResult with lcos::async
- **Issue #242**[^242] - REGISTER_TEMPLATE support for plain actions
- **Issue #243**[^243] - handle_gid<> support for hpx::util::function
- **Issue #245**[^245] - _c_cache code throws an exception if the queried GID is not in the local cache
- **Issue #246**[^246] - Undefined references in dataflow/adaptive1d example

[^71]: https://github.com/STEllAR-GROUP/hpx/issues/71
[^105]: https://github.com/STEllAR-GROUP/hpx/issues/105
[^107]: https://github.com/STEllAR-GROUP/hpx/issues/107
[^110]: https://github.com/STEllAR-GROUP/hpx/issues/110
[^116]: https://github.com/STEllAR-GROUP/hpx/issues/116
[^162]: https://github.com/STEllAR-GROUP/hpx/issues/162
[^172]: https://github.com/STEllAR-GROUP/hpx/issues/172
[^178]: https://github.com/STEllAR-GROUP/hpx/issues/178
[^187]: https://github.com/STEllAR-GROUP/hpx/issues/187
[^202]: https://github.com/STEllAR-GROUP/hpx/issues/202
[^218]: https://github.com/STEllAR-GROUP/hpx/issues/218
[^231]: https://github.com/STEllAR-GROUP/hpx/issues/231
[^232]: https://github.com/STEllAR-GROUP/hpx/issues/232
[^236]: https://github.com/STEllAR-GROUP/hpx/issues/236
[^239]: https://github.com/STEllAR-GROUP/hpx/issues/239
[^240]: https://github.com/STEllAR-GROUP/hpx/issues/240
[^242]: https://github.com/STEllAR-GROUP/hpx/issues/242
[^243]: https://github.com/STEllAR-GROUP/hpx/issues/243
[^245]: https://github.com/STEllAR-GROUP/hpx/issues/245
[^246]: https://github.com/STEllAR-GROUP/hpx/issues/246
- Issue #252 - Problems configuring sheneos with CMake
- Issue #254 - Lifetime of components doesn’t end when client goes out of scope
- Issue #259 - CMake does not detect that MSVC10 has lambdas
- Issue #260 - io_service_pool segfault
- Issue #261 - Late parcel executed outside of pxthread
- Issue #263 - Cannot select allocator with CMake
- Issue #264 - Fix allocator select
- Issue #267 - Runtime error for hello_world
- Issue #269 - pthread_affinity_np test fails to compile
- Issue #270 - Compiler noise due to -Wcast-qual
- Issue #275 - Problem with configuration tests/include paths on Gentoo
- Issue #325 - Sheneos is 200-400 times slower than the fortran equivalent
- Issue #331 - \texttt{hpx::init} and \texttt{hpx_main()} should not depend on program_options
- Issue #333 - Add doxygen support to CMake for doc toolchain
- Issue #340 - Performance counters for parcels
- Issue #346 - Component loading error when running hello_world in distributed on MSVC2010
- Issue #362 - Missing initializer error
- Issue #363 - Parcel port serialization error
- Issue #366 - Parcel buffering leads to types incompatible exception
- Issue #368 - Scalable alternative to rand() needed for HPX
- Issue #369 - IB over IP is substantially slower than just using standard TCP/IP
- Issue #374 - \texttt{hpx::lcos::wait} should work with dataflows and arbitrary classes meeting the future interface
- Issue #375 - Conflicting/ambiguous overloads of \texttt{hpx::lcos::wait}
• Issue #376 - Find_HPX.cmake should set CMake variable HPX_FOUND for out of tree builds
• Issue #377 - ShenEOS interpolate bulk and interpolate_one_bulk are broken
• Issue #379 - Add support for distributed runs under SLURM
• Issue #382 - _Unwind_Word not declared in boost.backtrace
• Issue #387 - Doxygen should look only at list of specified files
• Issue #388 - Running make install on an out-of-tree application is broken
• Issue #391 - Out-of-tree application segfaults when running in qsub
• Issue #392 - Remove HPX_NO_INSTALL option from cmake build system
• Issue #396 - Pragma related warnings when compiling with older gcc versions
• Issue #399 - Out of tree component build problems
• Issue #400 - Out of source builds on Windows: linker should not receive compiler flags
• Issue #401 - Out of source builds on Windows: components need to be linked with hpx_serialization
• Issue #404 - gfortran fails to link automatically when fortran files are present
• Issue #405 - Inability to specify linking order for external libraries
• Issue #406 - Adapt action limits such that dataflow applications work without additional defines
• Issue #415 - locality_results is not a member of hpx::components::server
• Issue #425 - Breaking changes to traits::*result wrt std::vector<id_type>
• Issue #426 - AUTOGLOB needs to be updated to support fortran

2.10.22 HPX V0.8.1 (Apr 21, 2012)

This is a point release including important bug fixes for HPX V0.8.0 (Mar 23, 2012).

https://github.com/STEllAR-GROUP/hpx/issues/376
https://github.com/STEllAR-GROUP/hpx/issues/377
https://github.com/STEllAR-GROUP/hpx/issues/379
https://github.com/STEllAR-GROUP/hpx/issues/382
https://github.com/STEllAR-GROUP/hpx/issues/387
https://github.com/STEllAR-GROUP/hpx/issues/388
https://github.com/STEllAR-GROUP/hpx/issues/391
https://github.com/STEllAR-GROUP/hpx/issues/392
https://github.com/STEllAR-GROUP/hpx/issues/396
https://github.com/STEllAR-GROUP/hpx/issues/399
https://github.com/STEllAR-GROUP/hpx/issues/400
https://github.com/STEllAR-GROUP/hpx/issues/401
https://github.com/STEllAR-GROUP/hpx/issues/404
https://github.com/STEllAR-GROUP/hpx/issues/405
https://github.com/STEllAR-GROUP/hpx/issues/406
https://github.com/STEllAR-GROUP/hpx/issues/415
https://github.com/STEllAR-GROUP/hpx/issues/425
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** - Don’t require install path to be known at compile time.
- **Issue #371** - Add hpx iostreams to standard build.
- **Issue #384** - Fix compilation with GCC 4.7.
- **Issue #390** - Remove keep_factory_alive startup call from ShenEOS; add shutdown call to H5close.
- **Issue #393** - 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.

2.10.23 *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.

---

4869 https://github.com/STEllAR-GROUP/hpx/issues/295
4870 https://github.com/STEllAR-GROUP/hpx/issues/371
4871 https://github.com/STEllAR-GROUP/hpx/issues/384
4872 https://github.com/STEllAR-GROUP/hpx/issues/390
4873 https://github.com/STEllAR-GROUP/hpx/issues/393

Chapter 2. What’s so special about *HPX*?
• 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[^4874] (thanks to Matthew Anderson for contributing this application).

• GTC (Gyrokinetic Toroidal Code)[^4875]: a skeleton for particle in cell type codes.

• Random Memory Access: an example demonstrating random memory accesses in a large array

• ShenEOS example[^4876], 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.

[^4874]: http://www.graph500.org/
[^4876]: http://stellarcollapse.org/equationofstate
Load balancing and work stealing demos.

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 4877 - Specialize handle_gid<> for examples and tests
- Issue #72 4878 - Fix AGAS reference counting
- Issue #104 4879 - heartbeat throws an exception when decrefing the performance counter it’s watching
- Issue #111 4880 - throttle causes an exception on the target application
- Issue #142 4881 - One failed component loading causes an unrelated component to fail
- Issue #165 4882 - Remote exception propagation bug in AGAS reference counting test
- Issue #186 4883 - Test credit exhaustion/splitting (e.g. prepare_id and symbol NS)
- Issue #188 4884 - Implement remaining AGAS reference counting test cases
- Issue #258 4885 - No type checking of GIDs in stubs classes

---

4877 https://github.com/STEllAR-GROUP/hpx/issues/31
4878 https://github.com/STEllAR-GROUP/hpx/issues/72
4879 https://github.com/STEllAR-GROUP/hpx/issues/104
4880 https://github.com/STEllAR-GROUP/hpx/issues/111
4881 https://github.com/STEllAR-GROUP/hpx/issues/142
4882 https://github.com/STEllAR-GROUP/hpx/issues/165
4883 https://github.com/STEllAR-GROUP/hpx/issues/186
4884 https://github.com/STEllAR-GROUP/hpx/issues/188
4885 https://github.com/STEllAR-GROUP/hpx/issues/258
• Issue #271 - Seg fault/shared pointer assertion in distributed code
• Issue #281 - CMake options need descriptive text
• Issue #283 - AGAS caching broken (gva_cache needs to be rewritten with ICL)
• Issue #285 - HPX_INSTALL root directory not the same as CMAKE_INSTALL_PREFIX
• Issue #286 - New segfault in dataflow applications
• Issue #289 - Exceptions should only be logged if not handled
• Issue #290 - C++11 tests failure
• Issue #293 - Build target for component libraries
• Issue #296 - Compilation error with Boost V1.49rc1
• Issue #298 - Illegal instructions on termination
• Issue #299 - Gravity aborts with multiple threads
• Issue #301 - Build error with Boost trunk
• Issue #303 - Logging assertion failure in distributed runs
• Issue #304 - Exception ‘what’ strings are lost when exceptions from decode_parcel are reported
• Issue #306 - Performance counter user interface issues
• Issue #307 - Logging exception in distributed runs
• Issue #308 - Logging deadlocks in distributed
• Issue #309 - Reference counting test failures and exceptions
• Issue #311 - Merge AGAS remote_interface with the runtime_support object
• Issue #314 - Object tracking for id_types
• Issue #315 - Remove handle_gid and handle credit splitting in id_type serialization
• Issue #320 - apply::get_locality_id() should return an error value (or throw an exception)
• Issue #321 - Optimization for id_types which are never split should be restored
• Issue #322\textsuperscript{4909} - Command line processing ignored with Boost 1.47.0
• Issue #323\textsuperscript{4910} - Credit exhaustion causes object to stay alive
• Issue #324\textsuperscript{4911} - Duplicate exception messages
• Issue #326\textsuperscript{4912} - Integrate Quickbook with CMake
• Issue #329\textsuperscript{4913} - --help and --version should still work
• Issue #330\textsuperscript{4914} - Create pkg-config files
• Issue #337\textsuperscript{4915} - Improve usability of performance counter timestamps
• Issue #338\textsuperscript{4916} - Non-std exceptions deriving from std::exceptions in tfunc may be sliced
• Issue #339\textsuperscript{4917} - Decrease the number of send_pending_parcel threads
• Issue #343\textsuperscript{4918} - Dynamically setting the stack size doesn’t work
• Issue #351\textsuperscript{4919} - ‘make install’ does not update documents
• Issue #353\textsuperscript{4920} - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs
• Issue #355\textsuperscript{4921} - ‘make’ doesn’t do anything after correct configuration
• Issue #356\textsuperscript{4922} - Don’t use `hpx::util::static_` in topology code
• Issue #359\textsuperscript{4923} - Infinite recursion in hpx::tuple serialization
• Issue #361\textsuperscript{4924} - Add compile time option to disable logging completely
• Issue #364\textsuperscript{4925} - Installation seriously broken in r7443

2.10.24 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.).

\textsuperscript{4909} https://github.com/STEllAR-GROUP/hpx/issues/322
\textsuperscript{4910} https://github.com/STEllAR-GROUP/hpx/issues/323
\textsuperscript{4911} https://github.com/STEllAR-GROUP/hpx/issues/324
\textsuperscript{4912} https://github.com/STEllAR-GROUP/hpx/issues/326
\textsuperscript{4913} https://github.com/STEllAR-GROUP/hpx/issues/329
\textsuperscript{4914} https://github.com/STEllAR-GROUP/hpx/issues/330
\textsuperscript{4915} https://github.com/STEllAR-GROUP/hpx/issues/337
\textsuperscript{4916} https://github.com/STEllAR-GROUP/hpx/issues/338
\textsuperscript{4917} https://github.com/STEllAR-GROUP/hpx/issues/339
\textsuperscript{4918} https://github.com/STEllAR-GROUP/hpx/issues/343
\textsuperscript{4919} https://github.com/STEllAR-GROUP/hpx/issues/351
\textsuperscript{4920} https://github.com/STEllAR-GROUP/hpx/issues/353
\textsuperscript{4921} https://github.com/STEllAR-GROUP/hpx/issues/355
\textsuperscript{4922} https://github.com/STEllAR-GROUP/hpx/issues/356
\textsuperscript{4923} https://github.com/STEllAR-GROUP/hpx/issues/359
\textsuperscript{4924} https://github.com/STEllAR-GROUP/hpx/issues/361
\textsuperscript{4925} 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 #28^4926 - Integrate Windows/Linux CMake code for HPX core
- Issue #32^4927 - hpx::cout() should be hpx::cout
- Issue #33^4928 - AGAS V2 legacy client does not properly handle error_code
- Issue #60^4929 - AGAS: allow for registerid to optionally take ownership of the gid
- Issue #62^4930 - adaptive1d compilation failure in Fusion
- Issue #64^4931 - Parcel subsystem doesn’t resolve domain names
- Issue #83^4932 - No error handling if no console is available
- Issue #84^4933 - No error handling if a hosted locality is treated as the bootstrap server
- Issue #90^4934 - Add general commandline option -N
- Issue #91^4935 - Add possibility to read command line arguments from file
- Issue #92^4936 - Always log exceptions/errors to the log file
- Issue #93^4937 - Log the command line/program name
- Issue #95^4938 - Support for distributed launches
- Issue #97^4939 - Attempt to create a bad component type in AMR examples
- Issue #100^4940 - factorial and factorial_get examples trigger AGAS component type assertions
- Issue #101^4941 - Segfault when hpx::process::here() is called in fibonacci2
- Issue #102^4942 - unknown_component_address in int_object_semaphore_client
- Issue #114^4943 - marduk raises assertion with default parameters
- Issue #115^4944 - Logging messages for SMP runs (on the console) shouldn’t be buffered

^4926 https://github.com/STEllAR-GROUP/hpx/issues/28
^4927 https://github.com/STEllAR-GROUP/hpx/issues/32
^4928 https://github.com/STEllAR-GROUP/hpx/issues/33
^4929 https://github.com/STEllAR-GROUP/hpx/issues/60
^4930 https://github.com/STEllAR-GROUP/hpx/issues/62
^4931 https://github.com/STEllAR-GROUP/hpx/issues/64
^4932 https://github.com/STEllAR-GROUP/hpx/issues/83
^4933 https://github.com/STEllAR-GROUP/hpx/issues/84
^4934 https://github.com/STEllAR-GROUP/hpx/issues/90
^4935 https://github.com/STEllAR-GROUP/hpx/issues/91
^4936 https://github.com/STEllAR-GROUP/hpx/issues/92
^4937 https://github.com/STEllAR-GROUP/hpx/issues/93
^4938 https://github.com/STEllAR-GROUP/hpx/issues/95
^4939 https://github.com/STEllAR-GROUP/hpx/issues/97
^4940 https://github.com/STEllAR-GROUP/hpx/issues/100
^4941 https://github.com/STEllAR-GROUP/hpx/issues/101
^4942 https://github.com/STEllAR-GROUP/hpx/issues/102
^4943 https://github.com/STEllAR-GROUP/hpx/issues/114
^4944 https://github.com/STEllAR-GROUP/hpx/issues/115
• Issue #119: marduk linking strategy breaks other applications
• Issue #121: pbsdsh problem
• Issue #123: marduk, dataflow and adaptive1d fail to build
• Issue #124: Lower default preprocessing arity
• Issue #125: Move hpx::detail::diagnostic_information out of the detail namespace
• Issue #126: Test definitions for AGAS reference counting
• Issue #128: Add averaging performance counter
• Issue #129: Error with endian.hpp while building adaptive1d
• Issue #130: Bad initialization of performance counters
• Issue #131: Add global startup/shutdown functions to component modules
• Issue #132: Avoid using auto_ptr
• Issue #133: On Windows hpx.dll doesn’t get installed
• Issue #134: HPX_LIBRARY does not reflect real library name (on Windows)
• Issue #135: Add detection of unique_ptr to build system
• Issue #137: Add command line option allowing to repeatedly evaluate performance counters
• Issue #139: Logging is broken
• Issue #140: CMake problem on windows
• Issue #141: Move all non-component libraries into $PREFIX/lib/hpx
• Issue #143: adaptive1d throws an exception with the default command line options
• Issue #146: Early exception handling is broken
• Issue #147: Sheneos doesn’t link on Linux
• Issue #149: sheneos_test hangs
• Issue #154: Compilation fails for r5661

https://github.com/STEllAR-GROUP/hpx/issues/119
https://github.com/STEllAR-GROUP/hpx/issues/121
https://github.com/STEllAR-GROUP/hpx/issues/123
https://github.com/STEllAR-GROUP/hpx/issues/124
https://github.com/STEllAR-GROUP/hpx/issues/125
https://github.com/STEllAR-GROUP/hpx/issues/126
https://github.com/STEllAR-GROUP/hpx/issues/128
https://github.com/STEllAR-GROUP/hpx/issues/129
https://github.com/STEllAR-GROUP/hpx/issues/130
https://github.com/STEllAR-GROUP/hpx/issues/131
https://github.com/STEllAR-GROUP/hpx/issues/132
https://github.com/STEllAR-GROUP/hpx/issues/133
https://github.com/STEllAR-GROUP/hpx/issues/134
https://github.com/STEllAR-GROUP/hpx/issues/135
https://github.com/STEllAR-GROUP/hpx/issues/136
https://github.com/STEllAR-GROUP/hpx/issues/137
https://github.com/STEllAR-GROUP/hpx/issues/138
https://github.com/STEllAR-GROUP/hpx/issues/139
https://github.com/STEllAR-GROUP/hpx/issues/140
https://github.com/STEllAR-GROUP/hpx/issues/141
https://github.com/STEllAR-GROUP/hpx/issues/143
https://github.com/STEllAR-GROUP/hpx/issues/146
https://github.com/STEllAR-GROUP/hpx/issues/147
https://github.com/STEllAR-GROUP/hpx/issues/149
https://github.com/STEllAR-GROUP/hpx/issues/154
• Issue #155\textsuperscript{4968} - Sine performance counters example chokes on chrono headers
• Issue #156\textsuperscript{4969} - Add build type to \texttt{--version}
• Issue #157\textsuperscript{4970} - Extend AGAS caching to store gid ranges
• Issue #158\textsuperscript{4971} - r5691 doesn’t compile
• Issue #160\textsuperscript{4972} - Re-add AGAS function for resolving a locality to its prefix
• Issue #168\textsuperscript{4973} - Managed components should be able to access their own GID
• Issue #169\textsuperscript{4974} - Rewrite AGAS future pool
• Issue #179\textsuperscript{4975} - Complete switch to request class for AGAS server interface
• Issue #182\textsuperscript{4976} - Sine performance counter is loaded by other examples
• Issue #185\textsuperscript{4977} - Write tests for symbol namespace reference counting
• Issue #191\textsuperscript{4978} - Assignment of read-only variable in point\_geometry
• Issue #200\textsuperscript{4979} - Seg faults when querying performance counters
• Issue #204\textsuperscript{4980} - \texttt{--ifnames} and suffix stripping needs to be more generic
• Issue #205\textsuperscript{4981} - \texttt{--list-*} and \texttt{--print-counter-*} options do not work together and produce no warning
• Issue #207\textsuperscript{4982} - Implement decrement entry merging
• Issue #208\textsuperscript{4983} - Replace the spinlocks in AGAS with \texttt{hpx::lcos::local_mutexes}
• Issue #210\textsuperscript{4984} - Add an \texttt{--ifprefix} option
• Issue #214\textsuperscript{4985} - Performance test for PX-thread creation
• Issue #216\textsuperscript{4986} - VS2010 compilation
• Issue #222\textsuperscript{4987} - r6045 context\_linux\_x86.hpp
• Issue #223\textsuperscript{4988} - fibonacci hangs when changing the state of an active thread
• Issue #225\textsuperscript{4989} - Active threads end up in the FEB wait queue
• Issue #226\textsuperscript{4990} - VS Build Error for Accumulator Client

\textsuperscript{4968} https://github.com/STEllAR-GROUP/hpx/issues/155
\textsuperscript{4969} https://github.com/STEllAR-GROUP/hpx/issues/156
\textsuperscript{4970} https://github.com/STEllAR-GROUP/hpx/issues/157
\textsuperscript{4971} https://github.com/STEllAR-GROUP/hpx/issues/158
\textsuperscript{4972} https://github.com/STEllAR-GROUP/hpx/issues/159
\textsuperscript{4973} https://github.com/STEllAR-GROUP/hpx/issues/160
\textsuperscript{4974} https://github.com/STEllAR-GROUP/hpx/issues/161
\textsuperscript{4975} https://github.com/STEllAR-GROUP/hpx/issues/162
\textsuperscript{4976} https://github.com/STEllAR-GROUP/hpx/issues/163
\textsuperscript{4977} https://github.com/STEllAR-GROUP/hpx/issues/164
\textsuperscript{4978} https://github.com/STEllAR-GROUP/hpx/issues/165
\textsuperscript{4979} https://github.com/STEllAR-GROUP/hpx/issues/166
\textsuperscript{4980} https://github.com/STEllAR-GROUP/hpx/issues/167
\textsuperscript{4981} https://github.com/STEllAR-GROUP/hpx/issues/168
\textsuperscript{4982} https://github.com/STEllAR-GROUP/hpx/issues/169
\textsuperscript{4983} https://github.com/STEllAR-GROUP/hpx/issues/170
\textsuperscript{4984} https://github.com/STEllAR-GROUP/hpx/issues/171
\textsuperscript{4985} https://github.com/STEllAR-GROUP/hpx/issues/172
\textsuperscript{4986} https://github.com/STEllAR-GROUP/hpx/issues/173
\textsuperscript{4987} https://github.com/STEllAR-GROUP/hpx/issues/174
\textsuperscript{4988} https://github.com/STEllAR-GROUP/hpx/issues/175
\textsuperscript{4989} https://github.com/STEllAR-GROUP/hpx/issues/176
\textsuperscript{4990} https://github.com/STEllAR-GROUP/hpx/issues/177
• Issue #228 - Move all traits into namespace hpx::traits
• Issue #229 - Invalid initialization of reference in thread_init_data
• Issue #235 - Invalid GID in iostreams
• Issue #238 - Demangle type names for the default implementation of get_action_name
• Issue #241 - C++11 support breaks GCC 4.5
• Issue #247 - Reference to temporary with GCC 4.4
• Issue #248 - Seg fault at shutdown with GCC 4.4
• Issue #253 - Default component action registration kills compiler
• Issue #272 - G++ unrecognized command line option
• Issue #273 - Quicksort example doesn’t compile
• Issue #277 - Invalid CMake logic for Windows

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:5002. Use the Zenodo entry for referring to the latest version of HPX:5003. Entries for citing specific versions of HPX can also be found at5004.

2.12 HPX users

A list of institutions and projects using HPX can be found on the HPX Users5005 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)5006, a multi-disciplinary research institute at Louisiana State University (LSU)5007. The ParalleX group was working to develop a new and
HPX Documentation, master

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 STEllAR group (Systems Technology, Emergent Parallelism, and Algorithm Research). Over time, the team grew to include researchers around the country and the world. In 2014, the STEllAR 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 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 STEllAR and ParalleX, see People and Why HPX?.

2.13.2 People

The STEllAR Group (pronounced as stellar) stands for “Systems Technology, Emergent Parallelism, and Algorithm Research”. We are an international group of faculty, researchers, and students working at various institutions around the world. The goal of the STEllAR 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.

**HPX contributors**

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartmut Kaiser</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:hkaiser@cct.lsu.edu">hkaiser@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Thomas Heller</td>
<td>Department of Computer Science 3 - Computer Architecture, Friedrich-Alexander University Erlangen-Nuremberg (FAU)</td>
<td><a href="mailto:thom.heller@gmail.com">thom.heller@gmail.com</a></td>
</tr>
<tr>
<td>Agustin Berge</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:kaballo86@hotmail.com">kaballo86@hotmail.com</a></td>
</tr>
<tr>
<td>Mikael Simberg</td>
<td>Swiss National Supercomputing Centre</td>
<td><a href="mailto:simbergm@cscs.ch">simbergm@cscs.ch</a></td>
</tr>
<tr>
<td>John Biddiscombe</td>
<td>Swiss National Supercomputing Centre</td>
<td><a href="mailto:biddisco@cscs.ch">biddisco@cscs.ch</a></td>
</tr>
<tr>
<td>Anton Bikineev</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:ant.bikineev@gmail.com">ant.bikineev@gmail.com</a></td>
</tr>
<tr>
<td>Martin Stumpf</td>
<td>Department of Computer Science 3 - Computer Architecture, Friedrich-Alexander University Erlangen-Nuremberg (FAU)</td>
<td><a href="mailto:martin.h.stumpf@gmail.com">martin.h.stumpf@gmail.com</a></td>
</tr>
<tr>
<td>Bryce Adelstein Lelbach</td>
<td>NVIDIA</td>
<td><a href="mailto:brycelelbach@gmail.com">brycelelbach@gmail.com</a></td>
</tr>
<tr>
<td>Shuangyang Yang</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:syang16@cct.lsu.edu">syang16@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Jeroen Habraken</td>
<td>Technische Universiteit Eindhoven</td>
<td><a href="mailto:jhabraken@cct.lsu.edu">jhabraken@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Steven Brandt</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:sbrandt@cct.lsu.edu">sbrandt@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Antoine Tran Tan</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:antoine.trantan@iri.fr">antoine.trantan@iri.fr</a></td>
</tr>
<tr>
<td>Adrian Serio</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:aserio@cct.lsu.edu">aserio@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Maciej Brodowicz</td>
<td>Center for Research in Extreme Scale Technologies (CREST), Indiana University (IU)</td>
<td><a href="mailto:mbrodowi@indiana.edu">mbrodowi@indiana.edu</a></td>
</tr>
</tbody>
</table>
## Contributors to this document

Table 2.36: Documentation authors

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartmut Kaiser</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:hkaiser@cct.lsu.edu">hkaiser@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Thomas Heller</td>
<td>Department of Computer Science 3 - Computer Architecture, Friedrich-Alexander University Erlangen-Nuremberg (FAU)</td>
<td><a href="mailto:thom.heller@gmail.com">thom.heller@gmail.com</a></td>
</tr>
<tr>
<td>Bryce Adelstein Leibach</td>
<td>NVIDIA</td>
<td><a href="mailto:brycelelbach@gmail.com">brycelelbach@gmail.com</a></td>
</tr>
<tr>
<td>Vinay C Amatya</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:vamaty@cct.lsu.edu">vamaty@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Steven Brandt</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:sbrandt@cct.lsu.edu">sbrandt@cct.lsu.edu</a></td>
</tr>
<tr>
<td>Maciej Brodowicz</td>
<td>Center for Research in Extreme Scale Technologies (CREST), Indiana University (IU)</td>
<td><a href="mailto:mbrodowl@indiana.edu">mbrodowl@indiana.edu</a></td>
</tr>
<tr>
<td>Adrian Serio</td>
<td>Center for Computation and Technology (CCT), Louisiana State University (LSU)</td>
<td><a href="mailto:aserio@cct.lsu.edu">aserio@cct.lsu.edu</a></td>
</tr>
</tbody>
</table>

---

5014 https://www.cct.lsu.edu  
5015 https://www.lsu.edu  
5016 https://www3.cs.fau.de  
5017 https://www.fau.de  
5018 https://www.cct.lsu.edu  
5019 https://www.lsu.edu  
5020 https://www.cs.cscs.ch  
5021 https://www.cs.cscs.ch  
5022 https://www.cct.lsu.edu  
5023 https://www.lsu.edu  
5024 https://www3.cs.fau.de  
5025 https://www.fau.de  
5026 https://nvidia.com/  
5027 https://www.cct.lsu.edu  
5028 https://www.lsu.edu  
5029 https://www.tui.nl  
5030 https://www.cct.lsu.edu  
5031 https://www.lsu.edu  
5032 https://www.cct.lsu.edu  
5033 https://www.lsu.edu  
5034 https://www.cct.lsu.edu  
5035 https://www.lsu.edu  
5036 https://pti.iu.edu  
5037 https://www.iu.edu  
5038 https://www.cct.lsu.edu  
5039 https://www.lsu.edu  
5040 https://www3.cs.fau.de  
5041 https://www.fau.de  
5042 https://nvidia.com/  
5043 https://www.cct.lsu.edu  
5044 https://www.lsu.edu  
5045 https://www.cct.lsu.edu  
5046 https://www.lsu.edu  
5047 https://pti.iu.edu  
5048 https://www.iu.edu  
5049 https://www.cct.lsu.edu  
5050 https://www.lsu.edu  

Chapter 2. What’s so special about HPX?
Acknowledgements

Thanks also to the following people who contributed directly or indirectly to the project through discussions, pull requests, documentation patches, etc.

- 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\(^{5051}\)), 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\(^{5052}\)), 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.
- Shoshana Jakobovits, for her work on the resource partitioner.

\(^{5051}\) https://www.cscs.ch
\(^{5052}\) https://www.cscs.ch
• 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.
• 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)\textsuperscript{5053}), 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)\textsuperscript{5054}), 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)\textsuperscript{5055}), 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 imple-
menting a HCC backend for HPX::Compute.
• Minh-Khanh Do, who contributed the implementation of several segmented algorithms.
• Bibek Wagle (Louisiana State University (LSU)\textsuperscript{5056}), 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\textsuperscript{5057}) with HPX. He reported various
problems and submitted several patches to fix issues allowing for a better integration with LibGeoDecomp\textsuperscript{5058}.
• 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)\textsuperscript{5059}), 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\textsuperscript{5060} C/C++ package manager.
• Vinay C Amatya (Louisiana State University (LSU)\textsuperscript{5061}), who contributed to the documentation and provided
some of the HPX examples.

\textsuperscript{5053} https://www.lsu.edu
\textsuperscript{5054} https://www.lsu.edu
\textsuperscript{5055} https://www.lsu.edu
\textsuperscript{5056} https://www.lsu.edu
\textsuperscript{5057} https://www.libgeodecomp.org/
\textsuperscript{5058} https://www.libgeodecomp.org/
\textsuperscript{5059} https://www.lsu.edu
\textsuperscript{5060} https://www.conan.io/
\textsuperscript{5061} https://www.lsu.edu
• Kevin Huck and Nick Chaimov (University of Oregon), 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.

• Patrick Diehl, who worked on implementing CUDA support for our companion library targeting GPGPUs (HPXCL).

• 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)) who contributed to HPX the implementation of several parallel algorithms (as proposed by N4313).

• Anuj Sharma and Christopher Bross (Department of Computer Science 3 - Computer Architecture), who worked on HPX in the context of the Google Summer of Code program 2014.

• Martin Stumpf (Department of Computer Science 3 - Computer Architecture), who rebuilt our contiguous testing infrastructure (see the HPX Buildbot Website). Martin is also working on HPXCL (mainly all work related to OpenCL) and implementing an HPX backend for POCL, a portable computing language solution based on OpenCL.

• Grant Mercer (University of Nevada, Las Vegas), who helped creating many of the parallel algorithms (as proposed by N4313).

• Damond Howard (Louisiana State University (LSU)), who works on HPXCL (mainly all work related to CUDA).

• 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. He also contributed an implementation of an API similar to Fortran co-arrays on top of HPX.

• John Biddiscombe (Swiss National Supercomputing Centre), 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.

---

5062 https://uoregon.edu/
5063 https://github.com/STEllAR-GROUP/hpxcl/
5064 https://www.cct.lsu.edu
5065 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
5066 https://www3.cs.fau.de
5067 https://developers.google.com/open-source/soc/
5068 http://rostam.cct.lsu.edu/
5069 https://github.com/STEllAR-GROUP/hpxcl/
5070 https://www.khronos.org/opencl/
5071 https://portablecl.org/
5072 https://www.unlv.edu
5073 https://www.khronos.org/opencl/
5074 https://www.lsu.edu
5075 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
5076 https://www.lsu.edu
5077 https://github.com/STEllAR-GROUP/hpxcl/
5079 https://www.numscale.com/nt2/
5080 https://www.cscs.ch

2.13. About HPX
• 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.

• Patricia Grubel (New Mexico State University), 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, Islada, 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 Dais who contributed to the general improvement of HPX.

HPX Funding Acknowledgements lists current and past funding sources for HPX.
• genindex