HPX Documentation

1.5.1

The STE||AR Group

October 01, 2020

USER DOCUMENTATION

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

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

- open an issue on GitHub[1];
- contact us on IRC, the HPX channel on the C++ Slack[2], or on our mailing list[3]; or
- read or ask questions tagged with *HPX* on StackOverflow[4].

See *Citing HPX* for details on how to cite *HPX* in publications.

[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

CHAPTER

ONE

WHAT IS *HPX*?

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

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

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

CHAPTER

TWO

WHAT'S SO SPECIAL ABOUT *HPX*?

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

2.1 Why *HPX*?

Current advances in high performance computing (HPC) continue to suffer from the issues plaguing parallel computation. These issues include, but are not limited to, ease of programming, inability to handle dynamically changing workloads, scalability, and efficient utilization of system resources. Emerging technological trends such as multi-core processors further highlight limitations of existing parallel computation models. To mitigate the aforementioned problems, it is necessary to rethink the approach to parallelization models. ParalleX contains mechanisms such as multi-threading, *parcels*, *global name space* support, percolation and *local control object*s (*LCO*). By design, ParalleX overcomes limitations of current models of parallelism by alleviating contention, latency, overhead and starvation. With ParalleX, it is further possible to increase performance by at least an order of magnitude on challenging parallel algorithms, e.g., dynamic directed graph algorithms and adaptive mesh refinement methods for astrophysics. An additional benefit of ParalleX is fine-grained control of power usage, enabling reductions in power consumption.

2.1.1 ParalleX—a new execution model for future architectures

ParalleX is a new parallel execution model that offers an alternative to the conventional computation models, such as message passing. ParalleX distinguishes itself by:

- Split-phase transaction model
- Message-driven
- Distributed shared memory (not cache coherent)
- Multi-threaded
- Futures synchronization
- *Local Control Object*s (*LCO*s)
- Synchronization for anonymous producer-consumer scenarios
- Percolation (pre-staging of task data)

The ParalleX model is intrinsically latency hiding, delivering an abundance of variable-grained parallelism within a hierarchical namespace environment. The goal of this innovative strategy is to enable future systems delivering very high efficiency, increased scalability and ease of programming. ParalleX can contribute to significant improvements in the design of all levels of computing systems and their usage from application algorithms and their programming languages to system architecture and hardware design together with their supporting compilers and operating system software.

2.1.2 What is *HPX*?

High Performance ParalleX (*HPX*) is the first runtime system implementation of the ParalleX execution model. The *HPX* runtime software package is a modular, feature-complete, and performance-oriented representation of the ParalleX execution model targeted at conventional parallel computing architectures, such as SMP nodes and commodity clusters. It is academically developed and freely available under an open source license. We provide *HPX* to the community for experimentation and application to achieve high efficiency and scalability for dynamic adaptive and irregular computational problems. *HPX* is a C++ library that supports a set of critical mechanisms for dynamic adaptive resource management and lightweight task scheduling within the context of a global address space. It is solidly based on many years of experience in writing highly parallel applications for HPC systems.

The two-decade success of the communicating sequential processes (CSP) execution model and its message passing interface (MPI) programming model have been seriously eroded by challenges of power, processor core complexity, multi-core sockets, and heterogeneous structures of GPUs. Both efficiency and scalability for some current (strong scaled) applications and future Exascale applications demand new techniques to expose new sources of algorithm parallelism and exploit unused resources through adaptive use of runtime information.

The ParalleX execution model replaces CSP to provide a new computing paradigm embodying the governing principles for organizing and conducting highly efficient scalable computations greatly exceeding the capabilities of today's problems. *HPX* is the first practical, reliable, and performance-oriented runtime system incorporating the principal concepts of the ParalleX model publicly provided in open source release form.

HPX is designed by the STE||AR[5] Group (**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch) at Louisiana State University (LSU)[6]'s Center for Computation and Technology (CCT)[7] to enable developers to exploit the full processing power of many-core systems with an unprecedented degree of parallelism. STE||AR[8] is a research group focusing on system software solutions and scientific application development for hybrid and many-core hardware architectures.

[5] https://stellar-group.org
[6] https://www.lsu.edu
[7] https://www.cct.lsu.edu
[8] https://stellar-group.org

For more information about the STE||AR[9] Group, see *People*.

2.1.3 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[10]—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[11]) and weak scaling (see Gustafson's Law[12]). Strong scaling is defined as how the solution time varies with the number of processors for a fixed **total** problem size. It gives an estimate of how much faster we can solve a particular problem by throwing more resources at it. Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size **per processor**. In other words, it defines how much more data can we process by using more hardware resources.

In order to utilize as much hardware parallelism as possible an application must exhibit excellent strong and weak scaling characteristics, which requires a high percentage of work executed in parallel, i.e., using multiple threads of execution. Optimally, if you execute an application on a hardware resource with N processors it either runs N times faster or it can handle N times more data. Both cases imply 100% of the work is executed on all available processors in parallel. However, this is just a theoretical limit. Unfortunately, there are more things that limit scalability, mostly inherent to the hardware architectures and the programming models we use. We break these limitations into four fundamental factors that make our systems *SLOW*:

- **S**tarvation occurs when there is insufficient concurrent work available to maintain high utilization of all resources.
- **L**atencies are imposed by the time-distance delay intrinsic to accessing remote resources and services.
- **O**verhead is work required for the management of parallel actions and resources on the critical execution path, which is not necessary in a sequential variant.
- **W**aiting for contention resolution is the delay due to the lack of availability of oversubscribed shared resources.

Each of those four factors manifests itself in multiple and different ways; each of the hardware architectures and programming models expose specific forms. However, the interesting part is that all of them are limiting the scalability of applications no matter what part of the hardware jungle we look at. Hand-helds, PCs, supercomputers, or the cloud, all suffer from the reign of the 4 horsemen: **S**tarvation, **L**atency, **O**verhead, and **C**ontention. This realization is very important as it allows us to derive the criteria for solutions to the scalability problem from first principles, and it allows us to focus our analysis on very concrete patterns and measurable metrics. Moreover, any derived results will be applicable to a wide variety of targets.

2.1.4 Technology demands new response

Today's computer systems are designed based on the initial ideas of John von Neumann[13], as published back in 1945, and later extended by the Harvard architecture[14]. 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:

[9] https://stellar-group.org
[10] http://en.wikipedia.org/wiki/FLOPS
[11] http://en.wikipedia.org/wiki/Amdahl%27s_law
[12] http://en.wikipedia.org/wiki/Gustafson%27s_law
[13] http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf
[14] http://en.wikipedia.org/wiki/Harvard_architecture

- 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[15]) 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.

2.1.5 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.

2.1.6 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[16], caching memory hierarchies in all modern processors, the constant optimization of existing MPI[17] implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use GPGPUs[18] 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.

[15] http://en.wikipedia.org/wiki/Mean_time_between_failures
[16] http://en.wikipedia.org/wiki/InfiniBand
[17] https://en.wikipedia.org/wiki/Message_Passing_Interface
[18] http://en.wikipedia.org/wiki/GPGPU

2.1.7 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[19]). 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)[20], Microsoft Parallel Patterns Library (PPL)[21], Cilk++[22], 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.

2.1.8 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[23]) or processes (MPI[24]). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP[25] as the system synchronizes the threads used to execute the different iterations in parallel. In MPI[26] 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.

2.1.9 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[27] (Partitioned Global Address Space) designed to over-

[19] http://en.wikipedia.org/wiki/Cray_XMT
[20] https://www.threadingbuildingblocks.org/
[21] https://msdn.microsoft.com/en-us/library/dd492418.aspx
[22] https://software.intel.com/en-us/articles/intel-cilk-plus/
[23] https://openmp.org/wp/
[24] https://en.wikipedia.org/wiki/Message_Passing_Interface
[25] https://openmp.org/wp/
[26] https://en.wikipedia.org/wiki/Message_Passing_Interface
[27] https://www.pgas.org/

come this limitation, such as Chapel[28], X10[29], UPC[30], or Fortress[31]. 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++[32]. The first attempts towards solving related problem go back decades as well, a good example is the Linda coordination language[33]. 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.

2.1.10 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[34] 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++[35]), we see better asynchrony, simpler application codes, and improved scaling.

2.1.11 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++[36] 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

[28] https://chapel.cray.com/
[29] https://x10-lang.org/
[30] https://upc.lbl.gov/
[31] https://labs.oracle.com/projects/plrg/Publications/index.html
[32] https://charm.cs.uiuc.edu/
[33] http://en.wikipedia.org/wiki/Linda_(coordination_language)
[34] http://en.wikipedia.org/wiki/GPGPU
[35] https://charm.cs.uiuc.edu/
[36] https://charm.cs.uiuc.edu/

the encoded action by passing along arguments and—possibly—continuations. *HPX* combines this scheme with work-queue based scheduling as described above, which allows the system to almost completely overlap any communication with useful work, thereby minimizing latencies.

2.2 Quick start

This section is intended to get you to the point of running a basic *HPX* program as quickly as possible. To that end we skip many details but instead give you hints and links to more details along the way.

We assume that you are on a Unix system with access to reasonably recent packages. You should have `cmake` and `make` available for the build system (`pkg-config` is also supported, see *Using HPX with pkg-config*).

2.2.1 Getting *HPX*

Download a tarball of the latest release from *HPX* Downloads[37] and unpack it or clone the repository directly using `git`:

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

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

```
git checkout 1.5.1
```

2.2.2 *HPX* dependencies

The minimum dependencies needed to use *HPX* are Boost[38] and Portable Hardware Locality (HWLOC)[39]. If these are not available through your system package manager, see *Installing Boost* and *Installing Hwloc* for instructions on how to build them yourself. In addition to Boost and Portable Hardware Locality (HWLOC), it is recommended that you don't use the system allocator, but instead use either `tcmalloc` from google-perftools[40] (default) or jemalloc[41] for better performance. If you would like to try *HPX* without a custom allocator at this point, you can configure *HPX* to use the system allocator in the next step.

A full list of required and optional dependencies, including recommended versions, is available at *Prerequisites*.

2.2.3 Building *HPX*

Once you have the source code and the dependencies, set up a separate build directory and configure the project. Assuming all your dependencies are in paths known to CMake, the following gets you started:

```
# In the HPX source directory
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/install/path ..
make install
```

[37] https://stellar-group.org/downloads/
[38] https://www.boost.org/
[39] https://www.open-mpi.org/projects/hwloc/
[40] https://code.google.com/p/gperftools
[41] http://jemalloc.net

This will build the core *HPX* libraries and examples, and install them to your chosen location. If you want to install *HPX* to system folders, simply leave out the `CMAKE_INSTALL_PREFIX` option. This may take a while. To speed up the process, launch more jobs by passing the `-jN` option to `make`.

Tip: Do not set only `-j` (i.e. `-j` without an explicit number of jobs) unless you have a lot of memory available on your machine.

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 `Release` and `Debug` modes.

If your dependencies are in custom locations, you may need to tell CMake where to find them by passing one or more of the following options to CMake:

```
-DBOOST_ROOT=/path/to/boost
-DHWLOC_ROOT=/path/to/hwloc
-DTCMALLOC_ROOT=/path/to/tcmalloc
-DJEMALLOC_ROOT=/path/to/jemalloc
```

If you want to try *HPX* without using a custom allocator pass `-DHPX_WITH_MALLOC=system` to CMake.

Important: If you are building *HPX* for a system with more than 64 processing units, you must change the CMake variables `HPX_WITH_MORE_THAN_64_THREADS` (to `On`) and `HPX_WITH_MAX_CPU_COUNT` (to a value at least as big as the number of (virtual) cores on your system).

To build the tests, run `make tests`. To run the tests, run either `make test` or use `ctest` for more control over which tests to run. You can run single tests for example with `ctest --output-on-failure -R tests.unit.parallel.algorithms.for_loop` or a whole group of tests with `ctest --output-on-failure -R tests.unit`.

If you did not run `make install` earlier, do so now or build the `hello_world_1` example by running:

```
make hello_world_1
```

HPX executables end up in the `bin` directory in your build directory. You can now run `hello_world_1` and should see the following output:

```
./bin/hello_world_1
Hello World!
```

You've just run an example which prints `Hello World!` from the *HPX* runtime. The source for the example is in `examples/quickstart/hello_world_1.cpp`. The `hello_world_distributed` example (also available in the `examples/quickstart` directory) is a distributed hello world program, which is described in *Remote execution with actions: Hello world*. It provides a gentle introduction to the distributed aspects of *HPX*.

Tip: Most build targets in *HPX* have two names: a simple name and a hierarchical name corresponding to what type of example or test the target is. If you are developing *HPX* it is often helpful to run `make help` to get a list of available targets. For example, `make help | grep hello_world` outputs the following:

```
... examples.quickstart.hello_world_2
... hello_world_2
```

(continues on next page)

(continued from previous page)

```
... examples.quickstart.hello_world_1
... hello_world_1
... examples.quickstart.hello_world_distributed
... hello_world_distributed
```

It is also possible to build, for instance, all quickstart examples using `make examples.quickstart`.

2.2.4 Installing and building *HPX* via vcpkg

You can download and install *HPX* using the *vcpkg <https://github.com/Microsoft/vcpkg>* dependency manager:

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg integrate install
vcpkg install hpx
```

The *HPX* port in vcpkg is kept up to date by Microsoft team members and community contributors. If the version is out of date, please *create an issue or pull request <https://github.com/Microsoft/vcpkg>* on the vcpkg repository.

2.2.5 Hello, World!

The following `CMakeLists.txt` is a minimal example of what you need in order to build an executable using CMake and *HPX*:

```
cmake_minimum_required(VERSION 3.13)
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)
```

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`.

Note: `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.

Note: `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.

Create a new project directory and a `CMakeLists.txt` with the contents above. Also create a `main.cpp` with the contents below.

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

(continues on next page)

(continued from previous page)

```
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
```

The program looks almost like a regular C++ hello world with the exception of the two includes and `hpx::cout`. When you include `hpx_main.hpp` some things will be done behind the scenes to make sure that `main` actually gets launched on the *HPX* runtime. So while it looks almost the same you can now use futures, `async`, parallel algorithms and more which make use of the *HPX* runtime with lightweight threads. `hpx::cout` is a replacement for `std::cout` to make sure printing never blocks a lightweight thread. You can read more about `hpx::cout` in *The HPX I/O-streams component*. If you rebuild and run your program now, you should see the familiar `Hello World!`:

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

Note: You do not have to let *HPX* take over your main function like in the example. You can instead keep your normal main function, and define a separate `hpx_main` function which acts as the entry point to the *HPX* runtime. In that case you start the *HPX* runtime explicitly by calling `hpx::init`:

```
//  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 initialize the HPX runtime explicitly and
// execute a HPX-thread printing "Hello World!" once. That's all.

//[hello_world_2_getting_started
#include <hpx/hpx_init.hpp>
#include <hpx/iostream.hpp>

int hpx_main(int, char**)
{
    // Say hello to the world!
    hpx::cout << "Hello World!\n" << hpx::flush;
    return hpx::finalize();
}

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

You can also use *hpx::start* and *hpx::stop* for a non-blocking alternative, or use *hpx::resume* and *hpx::suspend* if you need to combine *HPX* with other runtimes.

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.

2.2.6 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*:

```
#include <hpx/hpx_main.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/include/parallel_generate.hpp>
#include <hpx/include/parallel_sort.hpp>
#include <hpx/iostream.hpp>

#include <random>
#include <vector>

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

(continues on next page)

(continued from previous page)

```
// Avoid ABI incompatibilities between C++11/C++17 as std::rand has exception
// specification in libstdc++.
int rand_wrapper()
{
    return std::rand();
}

int main(int, char**)
{
    // 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; });
    hpx::cout << "Just launched a task!" << hpx::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.
    hpx::cout << "f contains " << f.get() << hpx::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.
    hpx::cout << "Result is ready? " << result.is_ready() << hpx::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::parallel::execution::seq, std::begin(v), std::end(v),
        &rand_wrapper);

    // We can launch the sort in parallel and asynchronously.
    hpx::future<void> done_sorting = hpx::parallel::sort(
        hpx::parallel::execution::par(          // In parallel.
            hpx::parallel::execution::task),    // Asynchronously.
        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.
    hpx::cout << (all.is_ready() ? "all is ready!" : "all is not ready...")
              << hpx::endl;
```

(continues on next page)

(continued from previous page)

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

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: Interest calculator* 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.

2.2.7 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: 1D stencil* 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.

2.3 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[42] models. Partitioned Global Address Space is passive in their means of address

[42] https://www.pgas.org/

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 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[43], the J-machine's[44] 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*.

[43] http://en.wikipedia.org/wiki/Dataflow_architecture

[44] http://en.wikipedia.org/wiki/J%E2%80%93Machine

2.4 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.

2.4.1 Asynchronous execution with `hpx::async`: Fibonacci

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.

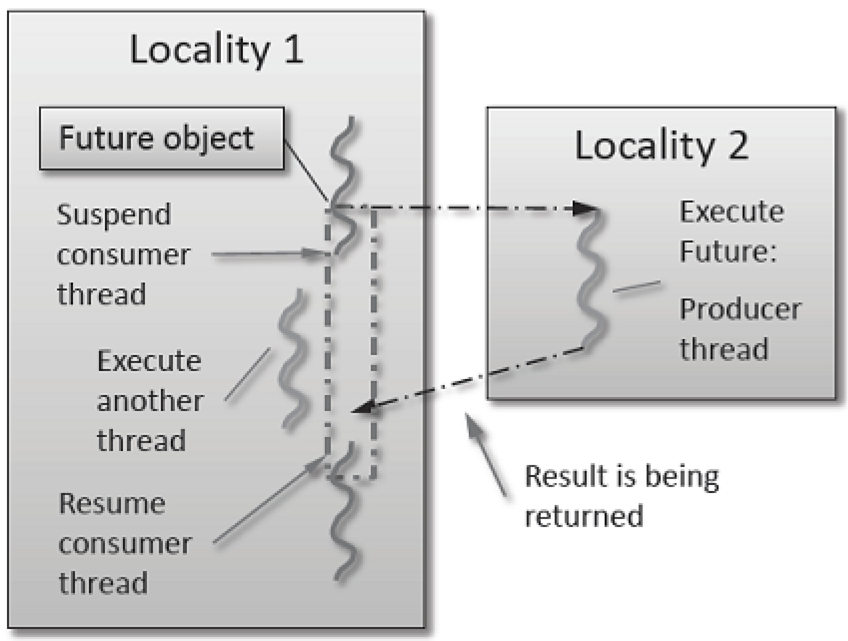

Fig. 2.1: Schematic of a future execution.

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:

```
./bin/fibonacci_local
```

This should print (time should be approximate):

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

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

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

Will yield:

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

Walkthrough

Now that you have compiled and run the code, let's look at how the code works. Since this code is written in C++, we will begin with the `main()` function. Here you can see that in *HPX*, `main()` is only used to initialize the runtime system. It is important to note that application-specific command line options are defined here. *HPX* uses Boost.Program Options[45] 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[46] for more details). `hpx::init` calls `hpx_main()` after setting up *HPX*, which is where the logic of our program is encoded.

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

(continues on next page)

[45] https://www.boost.org/doc/html/program_options.html
[46] https://www.boost.org/doc/

(continued from previous page)

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

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

```
int hpx_main(hpx::program_options::variables_map& vm)
{
    // 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::util::high_resolution_timer t;

        std::uint64_t r = fibonacci(n);

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

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

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

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

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

2.4.2 Asynchronous execution with `hpx::async` and actions: Fibonacci

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: Hello world*).

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*:

```
//[fib_main
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
    return hpx::init(desc_commandline, argc, argv);
```

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::util::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.

```
//[fib_hpx_main
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::util::high_resolution_timer t;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
// This is to generate the required boilerplate we need for the remote
// invocation to work.
```

A plain *action* is the most basic form of *action*. Plain *action*s wrap simple global functions which are not associated with any particular object (we will discuss other types of *actions* in *Components and actions: Accumulator*). 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()`:

```
//[fib_func
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.

2.4.3 Remote execution with actions: Hello world

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
```

This should print:

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

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

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

will yield:

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

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

Walkthrough

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

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

(continues on next page)

(continued from previous page)

```
    // 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 hpx::async and actions: Fibonacci* 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 hpx::async and actions: Fibonacci* our futures are set using *hpx::async<>*. The `hello_world_foreman_action` is declared here:

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
            // thread, but no guarantees are given by the scheduler that the
            // task will actually run on that worker thread.
            hpx::parallel::execution::default_executor exec(
                hpx::threads::thread_schedule_hint(
                    hpx::threads::thread_schedule_hint_mode_thread, 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::util::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`.

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

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

        return desired;
    }

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

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

2.4.4 Components and actions: Accumulator

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 hpx::async and actions: Fibonacci* and the *Remote execution with actions: Hello world*, 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
```

(continues on next page)

(continued from previous page)

```
1
> quit
```

Walkthrough

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

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

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

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

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

The server class

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

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

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

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

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

```
        argument_type value_;
```

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

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

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

```
        /// Reset the components value to 0.
        void reset()
        {
            //  set value_ to 0.
```

(continues on next page)

(continued from previous page)

```
            value_ = 0;
        }

        /// Add the given number to the accumulator.
        void add(argument_type arg)
        {
            //  add value_ to arg, and store the result in value_.
            value_ += arg;
        }

        /// Return the current value to the caller.
        argument_type query() const
        {
            // Get the value of value_.
            return value_;
        }
```

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 hpx::async and actions: Fibonacci* and the *Remote execution with actions: Hello world*:

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

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

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

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

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

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

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

```
//[accumulator_registration_definitions
///////////////////////////////////////////////////////////////////////////////
// Add factory registration functionality.
HPX_REGISTER_COMPONENT_MODULE();

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

HPX_REGISTER_COMPONENT(accumulator_type, accumulator);
```

(continues on next page)

(continued from previous page)

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

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

The client class

The following code is from `accumulator.hpp`.

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

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

and to invoke component actions:

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

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

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

For readability, we typedef the base class like so:

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

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

There are a few different ways of invoking actions:

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

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

            typedef server::accumulator::reset_action action_type;
```

(continues on next page)

(continued from previous page)

```
        hpx::apply<action_type>(this->get_id());
    }
```

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

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

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

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

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

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

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

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

2.4.5 Dataflow: Interest calculator

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
```

This should print:

```
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 hpx::async and actions: Fibonacci* 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.

```
///////////////////////////////////////////////////////////////////////////////
//[interest_main
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]")
    ;
```

Next we look at hpx_main.

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

(continues on next page)

(continued from previous page)

```
        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;
    }
```

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`:

```
///////////////////////////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

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

///////////////////////////////////////////////////////////////////////////////
// Add the amount made to the principal
double add(double principal, double interest)
{
```

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

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

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

2.4.6 Local to remote: 1D stencil

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.

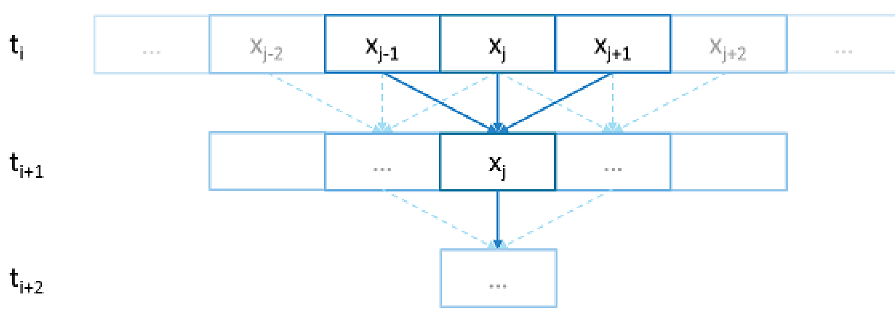

Fig. 2.2: Heat diffusion example program flow.

We parallelize this code over the following eight examples:

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

- `Example 7`
- `Example 8`

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

```
//[stepper_1
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`.

```
//[stepper_2
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::util::unwrapping;

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
        // 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 grainsize of the simulation. The rest of the code is then futurized in the same manner as example 2. It should be noted how strikingly similar example 4 is to example 2.

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

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

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

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

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

```
// Initial conditions: f(0, i) = i
//[do_work_6
```

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:

```
//[stepper_7
// 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::util::unwrapping;

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

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

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

(continues on next page)

(continued from previous page)

```
                    next[i] = heat(m[i-1], m[i], m[i+1]);
                return next;
            }
        )
    );

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

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

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

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

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

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

2.5 Manual

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

2.5.1 Getting *HPX*

There are *HPX* packages available for a few Linux distributions. The easiest way to get started with *HPX* is to use those packages. We keep an up-to-date list with instructions on the *HPX* Downloads[47] page. If you use one of the available packages you can skip the next section, *HPX build system*, but we still recommend that you look through it as it contains useful information on how you can customize *HPX* at compile-time.

If there isn't a package available for your platform you should either clone our repository:

or download a package with the source files from *HPX* Downloads[48].

2.5.2 *HPX* build system

The build system for *HPX* is based on CMake[49]. CMake is a cross-platform build-generator tool. CMake does not build the project, it generates the files needed by your build tool (GNU make, Visual Studio, etc.) for building *HPX*.

This section gives an introduction on how to use our build system to build *HPX* and how to use *HPX* in your own projects.

CMake basics

CMake is a cross-platform build-generator tool. CMake does not build the project, it generates the files needed by your build tool (gnu make, visual studio, etc.) for building *HPX*.

In general, the *HPX* CMake scripts try to adhere to the general CMake policies on how to write CMake-based projects.

Basic CMake usage

This section explains basic aspects of CMake, specifically options needed for day-to-day usage.

CMake comes with extensive documentation in the form of html files and on the CMake executable itself. Execute `cmake --help` for further help options.

CMake needs to know which build tool it will generate files for (GNU make, Visual Studio, Xcode, etc.). If not specified on the command line, it will try to guess the build tool based on you environment. Once it has identified the build tool, CMake uses the corresponding generator to create files for your build tool. You can explicitly specify the generator with the command line option `-G "Name of the generator"`. To see the available generators on your platform, execute:

```
cmake --help
```

This will list the generator names at the end of the help text. Generator names are case-sensitive. Example:

```
cmake -G "Visual Studio 16 2019" path/to/hpx
```

47 https://stellar-group.org/downloads/
48 https://stellar-group.org/downloads/
49 https://www.cmake.org

For a given development platform there can be more than one adequate generator. If you use Visual Studio "NMake Makefiles" is a generator you can use for building with NMake. By default, CMake chooses the more specific generator supported by your development environment. If you want an alternative generator, you must tell this to CMake with the -G option.

Quick start

Here, you will use the command-line, non-interactive CMake interface.

1. Download and install CMake here: CMake Downloads[50]. Version 3.13 is the minimum required version for *HPX*.
2. Open a shell. Your development tools must be reachable from this shell through the PATH environment variable.
3. Create a directory for containing the build. Building *HPX* on the source directory is not supported. cd to this directory:

```
mkdir mybuilddir
cd mybuilddir
```

4. Execute this command on the shell replacing path/to/hpx with the path to the root of your *HPX* source tree:

```
cmake path/to/hpx
```

CMake will detect your development environment, perform a series of tests and will generate the files required for building *HPX*. CMake will use default values for all build parameters. See the *CMake variables used to configure HPX* section for fine-tuning your build.

This can fail if CMake can't detect your toolset, or if it thinks that the environment is not sane enough. In this case make sure that the toolset that you intend to use is the only one reachable from the shell and that the shell itself is the correct one for you development environment. CMake will refuse to build MinGW makefiles if you have a POSIX shell reachable through the PATH environment variable, for instance. You can force CMake to use various compilers and tools. Please visit CMake Useful Variables[51] for a detailed overview of specific CMake variables.

Options and variables

Variables customize how the build will be generated. Options are boolean variables, with possible values ON/OFF. Options and variables are defined on the CMake command line like this:

```
cmake -DVARIABLE=value path/to/hpx
```

You can set a variable after the initial CMake invocation for changing its value. You can also undefine a variable:

```
cmake -UVARIABLE path/to/hpx
```

Variables are stored on the CMake cache. This is a file named CMakeCache.txt on the root of the build directory. Do not hand-edit it.

Variables are listed here appending its type after a colon. You should write the variable and the type on the CMake command line:

```
cmake -DVARIABLE:TYPE=value path/to/llvm/source
```

[50] https://www.cmake.org/cmake/resources/software.html
[51] https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/Useful-Variables#Compilers-and-Tools

CMake supports the following variable types: `BOOL` (options), `STRING` (arbitrary string), `PATH` (directory name), `FILEPATH` (file name).

Prerequisites

Supported platforms

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

Table 2.1: Supported Platforms for *HPX*

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

Software and libraries

In the simplest case, *HPX* depends on Boost[52] and Portable Hardware Locality (HWLOC)[53]. So, before you read further, please make sure you have a recent version of Boost[54] installed on your target machine. *HPX* currently requires at least Boost V1.61.0 to work properly. It may build and run with older versions, but we do not test *HPX* with those versions, so please be warned.

The installation of Boost is described in detail in Boost's Getting Started[55] document. However, if you've never used the Boost libraries (or even if you have), here's a quick primer: *Installing Boost*.

It is often possible to download the Boost libraries using the package manager of your distribution. Please refer to the corresponding documentation for your system for more information.

In addition, we require a recent version of hwloc in order to support thread pinning and NUMA awareness. See *Installing Hwloc* for instructions on building Portable Hardware Locality (HWLOC).

HPX is written in 99.99% Standard C++ (the remaining 0.01% is platform specific assembly code). As such, *HPX* is compilable with almost any standards compliant C++ compiler. A compiler supporting the C++11 Standard is highly recommended. The code base takes advantage of C++11 language features when available (move semantics, rvalue references, magic statics, etc.). This may speed up the execution of your code significantly. We currently support the following C++ compilers: GCC, MSVC, ICPC and clang. For the status of your favorite compiler with *HPX* visit *HPX* Buildbot Website[56].

[52] https://www.boost.org/
[53] https://www.open-mpi.org/projects/hwloc/
[54] https://www.boost.org/
[55] https://www.boost.org/doc/libs/1_73_0/more/getting_started/index.html
[56] http://rostam.cct.lsu.edu/

Table 2.2: Software prerequisites for *HPX* on Linux systems.

Name	Minimum version	Notes
Compilers		
GNU Compiler Collection (g++)[57]	7.0	
Intel Composer XE Suites[58]	2014	
clang: a C language family frontend for LLVM[59]	5.0	
Build System		
CMake[60]	3.13	Cuda support 3.9
Required Libraries		
Boost C++ Libraries[61]	1.64.0	
Portable Hardware Locality (HWLOC)[62]	1.5	

Note: When building Boost using gcc, please note that it is required to specify a `cxxflags=-std=c++14` command line argument to `b2` (`bjam`).

Table 2.3: Software prerequisites for *HPX* on Windows systems

Name	Minimum version	Notes
Compilers		
Visual C++[63] (x64)	2015	
Build System		
CMake[64]	3.13	
Required Libraries		
Boost[65]	1.64.0	
Portable Hardware Locality (HWLOC)[66]	1.5	

Note: You need to build the following Boost libraries for *HPX*: Boost.Filesystem, Boost.ProgramOptions, and Boost.System. The following are not needed by default, but are required in certain configurations: Boost.Chrono, Boost.DateTime, Boost.Log, Boost.LogSetup, Boost.Regex, and Boost.Thread.

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

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

[57] https://gcc.gnu.org
[58] https://software.intel.com/en-us/intel-composer-xe/
[59] https://clang.llvm.org/
[60] https://www.cmake.org
[61] https://www.boost.org/
[62] https://www.open-mpi.org/projects/hwloc/
[63] https://msdn.microsoft.com/en-us/visualc/default.aspx
[64] https://www.cmake.org
[65] https://www.boost.org/
[66] https://www.open-mpi.org/projects/hwloc/

Table 2.4: Highly recommended optional software prerequisites for *HPX* on Linux systems

Name	Minimum version	Notes
google-perftools[67]	1.7.1	Used as a replacement for the system allocator, and for allocation diagnostics.
libunwind[68]	0.97	Dependency of google-perftools on x86-64, used for stack unwinding.
Open MPI[69]	1.8.0	Can be used as a highspeed communication library backend for the parcelport.

Note: When using OpenMPI please note that Ubuntu (notably 18.04 LTS) and older Debian ship an OpenMPI 2.x built with `--enable-heterogeneous` which may cause communication failures at runtime and should not be used.

Table 2.5: Optional software prerequisites for *HPX* on Linux systems

Name	Minimum version	Notes
Performance Application Programming Interface (PAPI)		Used for accessing hardware performance data.
jemalloc[70]	2.1.0	Used as a replacement for the system allocator.
mi-malloc[71]	1.0.0	Used as a replacement for the system allocator.
Hierarchical Data Format V5 (HDF5)[72]	1.6.7	Used for data I/O in some example applications. See important note below.

Table 2.6: Optional software prerequisites for *HPX* on Windows systems

Name	Minimum version	Notes
Hierarchical Data Format V5 (HDF5)[73]	1.6.7	Used for data I/O in some example applications. See important note below.

Important: The C++ HDF5 libraries must be compiled with enabled thread safety support. This has to be explicitly specified while configuring the HDF5 libraries as it is not the default. Additionally, you must set the following environment variables before configuring the HDF5 libraries (this part only needs to be done on Linux):

```
export CFLAGS='-DHDatexit=""'
export CPPFLAGS='-DHDatexit=""'
```

[67] https://code.google.com/p/gperftools
[68] https://www.nongnu.org/libunwind
[69] https://www.open-mpi.org
[70] http://jemalloc.net
[71] http://microsoft.github.io/mimalloc/
[72] https://www.hdfgroup.org/HDF5
[73] https://www.hdfgroup.org/HDF5

Documentation

To build the *HPX* documentation, you need recent versions of the following packages:

- `python3`
- `sphinx` (Python package)
- `sphinx_rtd_theme` (Python package)
- `breathe 4.16.0` (Python package)
- `doxygen`

If the Python[74] dependencies are not available through your system package manager, you can install them using the Python package manager `pip`:

```
pip install --user sphinx sphinx_rtd_theme breathe
```

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[75] tool.

SPHINX_ROOT:PATH
Specifies where to look for the installation of the Sphinx[76] tool.

BREATHE_APIDOC_ROOT:PATH
Specifies where to look for the installation of the Breathe[77] tool.

Installing Boost

Important: When building Boost using gcc, please note that it is required to specify a `cxxflags=-std=c++14` command line argument to `b2` (`bjam`).

Important: On Windows, depending on the installed versions of Visual Studio, you might also want to pass the correct toolset to the `b2` command depending on which version of the IDE you want to use. In addition, passing `address-model=64` is highly recommended. It might also be necessary to add command line argument `--build-type=complete` to the `b2` command on the Windows platform.

The easiest way to create a working Boost installation is to compile Boost from sources yourself. This is particularly important as many high performance resources, even if they have Boost installed, usually only provide you with an older version of Boost. We suggest you download the most recent release of the Boost libraries from here: Boost Downloads[78]. Unpack the downloaded archive into a directory of your choosing. We will refer to this directory a `$BOOST`.

Building and installing the Boost binaries is simple. Regardless of what platform you are on, the basic instructions are as follows (with possible additional platform-dependent command line arguments):

[74] https://www.python.org
[75] https://www.doxygen.org
[76] http://www.sphinx-doc.org
[77] https://breathe.readthedocs.io/en/latest
[78] https://www.boost.org/users/download/

```
cd $BOOST
bootstrap --prefix=<where to install boost>
./b2 -j<N>
./b2 install
```

where: `<where to install boost>` is the directory the built binaries will be installed to, and `<N>` is the number of cores to use to build the Boost binaries.

After the above sequence of commands has been executed (this may take a while!), you will need to specify the directory where Boost was installed as `BOOST_ROOT` (`<where to install boost>`) while executing CMake for *HPX* as explained in detail in the sections *How to install HPX on Unix variants* and *How to install HPX on Windows*.

Installing Hwloc

Note: These instructions are for everything except Windows. On Windows there is no need to build hwloc. Instead, download the latest release, extract the files, and set `HWLOC_ROOT` during CMake configuration to the directory in which you extracted the files.

We suggest you download the most recent release of hwloc from here: Hwloc Downloads[79]. Unpack the downloaded archive into a directory of your choosing. We will refer to this directory as `$HWLOC`.

To build hwloc run:

```
cd $HWLOC
./configure --prefix=<where to install hwloc>
make -j<N> install
```

where: `<where to install hwloc>` is the directory the built binaries will be installed to, and `<N>` is the number of cores to use to build hwloc.

After the above sequence of commands has been executed, you will need to specify the directory where hwloc was installed as `HWLOC_ROOT` (`<where to install hwloc>`) while executing CMake for *HPX* as explained in detail in the sections *How to install HPX on Unix variants* and *How to install HPX on Windows*.

Please see Hwloc Documentation[80] for more information about hwloc.

Building *HPX*

Basic information

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. The generated targets are:
  - `hpx`: The core *HPX* library (always enabled).
  - `hpx_init`: The *HPX* initialization library that applications need to link against to define the *HPX* entry points (disabled for static builds).

[79] https://www.open-mpi.org/software/hwloc/v1.11
[80] https://www.open-mpi.org/projects/hwloc/doc/

  - `hpx_wrap`: The *HPX* static library used to determine the runtime behavior of *HPX* code and respective entry points for `hpx_main.h`
  - `iostreams_component`: The component used for (distributed) IO (always enabled).
  - `component_storage_component`: The component needed for migration to persistent storage.
  - `unordered_component`: The component needed for a distributed (partitioned) hash table.
  - `partioned_vector_component`: The component needed for a distributed (partitioned) vector.
  - `memory_component`: A dynamically loaded plugin that exposes memory based performance counters (only available on Linux).
  - `io_counter_component`: A dynamically loaded plugin that exposes I/O performance counters (only available on Linux).
  - `papi_component`: A dynamically loaded plugin that exposes PAPI performance counters (enabled with *HPX_WITH_PAPI:BOOL*, default is `Off`).

- *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 (the ones marked as optional above are truly optional). When building against *HPX*, the CMake variable `HPX_LIBRARIES` will contain `hpx` and `hpx_init` (for pkgconfig, those are added to the `Libs` sections). 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, we require a certain minimum set of features from the C++11 standard. In addition, we make use of certain C++14 features if the used compiler supports them. This means that the *HPX* build system will try to determine the highest support C++ standard flavor and check for availability of those features. That is, the default will be the highest C++ standard version available. If you want to force *HPX* to use a specific C++ standard version, you can use the following CMake variables:

- `HPX_WITH_CXX14`: Enables C++14 support (this is the minimum requirement)
- `HPX_WITH_CXX17`: Enables C++17 support
- `HPX_WITH_CXX2A`: Enables (experimental) C++20 support

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 `CMAKE_BUILD_TYPE` (for more information see the CMake Documentation[81]). Available build types are:

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

Important: We currently don't guarantee ABI compatibility between Debug and Release builds. Please make sure that applications built against *HPX* use the same build type as you used to build *HPX*. For CMake builds, this means that the `CMAKE_BUILD_TYPE` variables have to match and for projects not using CMake[82], the `HPX_DEBUG` macro has to be set in debug mode.

Platform specific notes

Some platforms require users to have special link and/or compiler flags specified to build *HPX*. This is handled via CMake's support for different toolchains (see cmake-toolchains(7)[83] for more information). This is also used for cross compilation.

HPX ships with a set of toolchains that can be used for compilation of *HPX* itself and applications depending on *HPX*. Please see *CMake toolchains shipped with HPX* for more information.

In order to enable full static linking with the libraries, the CMake variable *`HPX_WITH_STATIC_LINKING:BOOL`* has to be set to `On`.

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:

[81] https://cmake.org/cmake/help/latest/variable/CMAKE_BUILD_TYPE.html
[82] https://www.cmake.org
[83] https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html

```
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`.

Platform specific build recipes

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

How to install *HPX* on Unix variants

- Create a build directory. *HPX* requires an out-of-tree build. This means you will be unable to run CMake in the *HPX* source tree.

```
cd hpx
mkdir my_hpx_build
cd my_hpx_build
```

- Invoke CMake from your build directory, pointing the CMake driver to the root of your *HPX* source tree.

```
cmake -DBOOST_ROOT=/root/of/boost/installation \
      -DHWLOC_ROOT=/root/of/hwloc/installation
      [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
```

- Invoke GNU make. If you are on a machine with multiple cores, add the -jN flag to your make invocation, where N is the number of parallel processes *HPX* gets compiled with.

```
gmake -j4
```

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.

Note: Many Linux distributions use `make` as an alias for `gmake`.

- To complete the build and install *HPX*:

```
gmake install
```

Important: These commands will build and install the essential core components of *HPX* only. In order to build and run the tests, please invoke:

```
gmake tests && gmake test
```

and in order to build (and install) all examples invoke:

```
cmake -DHPX_WITH_EXAMPLES=On .
gmake examples
gmake install
```

For more detailed information about using CMake, please refer to its documentation and also the section *Building HPX*. Please pay special attention to the section about *HPX_WITH_MALLOC:STRING* as this is crucial for getting decent performance.

How to install *HPX* on OS X (Mac)

This section describes how to build *HPX* for OS X (Mac).

Build (and install) a recent version of Boost, using Clang and libc++

To build Boost with Clang and make it link to libc++ as standard library, you'll need to set up either of the following in your `~/user-config.jam` file:

```
# user-config.jam (put this file into your home directory)
# ...

using clang
    :
    : "/usr/bin/clang++"
    : <cxxflags>"-std=c++11 -fcolor-diagnostics"
      <linkflags>"-stdlib=libc++ -L/path/to/libcxx/lib"
    ;
```

(Again, remember to replace `/path/to` with whatever you used earlier.)

Then, you can use one of the following for your build command:

```
b2 --build-dir=/tmp/build-boost --layout=versioned toolset=clang install -j4
```

or:

```
b2 --build-dir=/tmp/build-boost --layout=versioned toolset=clang install -j4
```

We verified this using Boost V1.53. If you use a different version, just remember to replace `/usr/local/include/boost-1_53` with whatever prefix you used in your installation.

Build *HPX*, finally

```
cd /path/to
git clone https://github.com/STEllAR-GROUP/hpx.git
mkdir build-hpx && cd build-hpx
```

To build with Clang, execute:

```
cmake ../hpx \
    -DCMAKE_CXX_COMPILER=clang++ \
    -DBOOST_ROOT=/path/to/boost \
    -DHWLOC_ROOT=/path/to/hwloc \
    -DHPX_WITH_GENERIC_CONTEXT_COROUTINES=On
make -j
```

For more detailed information about using CMake, please refer its documentation and to the section *Building HPX*.

Alternative installation method of *HPX* on OS X (Mac)

Alternatively, you can install a recent version of gcc as well as all required libraries via MacPorts:

1. Install MacPorts
2. Install CMake, gcc, hwloc:

```
sudo brew install cmake
sudo brew install boost
sudo brew install hwloc
sudo brew install make
```

3. You may also want:

```
sudo brew install gperftools
```

4. If you need to build Boost manually (the Boost package of MacPorts is built with Clang, and unfortunately doesn't work with a GCC-build version of *HPX*):

```
wget https://dl.bintray.com/boostorg/release/1.69.0/source/boost_1_69_0.tar.bz2
tar xjf boost_1_69_0.tar.bz2
pushd boost_1_69_0
export BOOST_ROOT=$HOME/boost_1_69_0
./bootstrap.sh --prefix=$BOOST_DIR
./b2 -j8
./b2 -j8 install
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:$BOOST_ROOT/lib
popd
```

5. Build *HPX*:

```
git clone https://github.com/STEllAR-GROUP/hpx.git
mkdir hpx-build
pushd hpx-build
export HPX_ROOT=$HOME/hpx
cmake -DCMAKE_C_COMPILER=gcc \
    -DCMAKE_CXX_COMPILER=g++ \
    -DCMAKE_FORTRAN_COMPILER=gfortran \
```

(continues on next page)

(continued from previous page)

```
    -DCMAKE_C_FLAGS="-Wno-unused-local-typedefs" \
    -DCMAKE_CXX_FLAGS="-Wno-unused-local-typedefs" \
    -DBOOST_ROOT=$BOOST_ROOT \
    -DHWLOC_ROOT=/opt/local \
    -DCMAKE_INSTALL_PREFIX=$HOME/hpx \
    -DHPX_WITH_GENERIC_CONTEXT_COROUTINES=On \
         $(pwd)/../hpx
make -j8
make -j8 install
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:$HPX_ROOT/lib/hpx
popd
```

6. Note that you need to set `BOOST_ROOT`, `HPX_ROOT` and `DYLD_LIBRARY_PATH` (for both `BOOST_ROOT` and `HPX_ROOT`) every time you configure, build, or run an *HPX* application.
7. Note that you need to set `HPX_WITH_GENERIC_CONTEXT_COROUTINES=On` for MacOS.
8. If you want to use *HPX* with MPI, you need to enable the MPI parcelport, and also specify the location of the MPI wrapper scripts. This can be done using the following command:

```
cmake -DHPX_WITH_PARCELPORT_MPI=ON \
     -DCMAKE_C_COMPILER=gcc \
     -DCMAKE_CXX_COMPILER=g++ \
     -DCMAKE_FORTRAN_COMPILER=gfortran \
     -DMPI_C_COMPILER=openmpicc \
     -DMPI_CXX_COMPILER=openmpic++ \
     -DMPI_FORTRAN_COMPILER=openmpif90 \
     -DCMAKE_C_FLAGS="-Wno-unused-local-typedefs" \
     -DCMAKE_CXX_FLAGS="-Wno-unused-local-typedefs" \
     -DBOOST_ROOT=$BOOST_DIR \
     -DHWLOC_ROOT=/opt/local \
     -DCMAKE_INSTALL_PREFIX=$HOME/hpx
         $(pwd)/../hpx
```

How to install *HPX* on Windows

Installation of required prerequisites

- Download the Boost c++ libraries from Boost Downloads[84]
- Install the Boost library as explained in the section *Installing Boost*
- Install the hwloc library as explained in the section *Installing Hwloc*
- Download the latest version of CMake binaries, which are located under the platform section of the downloads page at CMake Downloads[85].
- Download the latest version of *HPX* from the STE||AR website: *HPX* Downloads[86].

[84] https://www.boost.org/users/download/
[85] https://www.cmake.org/cmake/resources/software.html
[86] https://stellar-group.org/downloads/

Installation of the *HPX* library

- Create a build folder. *HPX* requires an out-of-tree-build. This means that you will be unable to run CMake in the *HPX* source folder.
- Open up the CMake GUI. In the input box labelled "Where is the source code:", enter the full path to the source folder. The source directory is the one where the sources were checked out. CMakeLists.txt files in the source directory as well as the subdirectories describe the build to CMake. In addition to this, there are CMake scripts (usually ending in .cmake) stored in a special CMake directory. CMake does not alter any file in the source directory and doesn't add new ones either. In the input box labelled "Where to build the binaries:", enter the full path to the build folder you created before. The build directory is one where all compiler outputs are stored, which includes object files and final executables.
- Add CMake variable definitions (if any) by clicking the "Add Entry" button. There are two required variables you need to define: `BOOST_ROOT` and `HWLOC_ROOT` These (`PATH`) variables need to be set to point to the root folder of your Boost and hwloc installations. It is recommended to set the variable `CMAKE_INSTALL_PREFIX` as well. This determines where the *HPX* libraries will be built and installed. If this (`PATH`) variable is set, it has to refer to the directory where the built *HPX* files should be installed to.
- Press the "Configure" button. A window will pop up asking you which compilers to use. Select the Visual Studio 10 (64Bit) compiler (it usually is the default if available). The Visual Studio 2012 (64Bit) and Visual Studio 2013 (64Bit) compilers are supported as well. Note that while it is possible to build *HPX* for x86, we don't recommend doing so as 32 bit runs are severely restricted by a 32 bit Windows system limitation affecting the number of *HPX* threads you can create.
- Press "Configure" again. Repeat this step until the "Generate" button becomes clickable (and until no variable definitions are marked in red anymore).
- Press "Generate".
- Open up the build folder, and double-click hpx.sln.
- Build the INSTALL target.

For more detailed information about using CMake[87] please refer its documentation and also the section *Building HPX*.

How to build *HPX* under Windows 10 x64 with Visual Studio 2015

- Download the CMake V3.18.1 installer (or latest version) from here[88]
- Download the hwloc V1.11.0 (or the latest version) from here[89] and unpack it.
- Download the latest Boost libraries from here[90] and unpack them.
- Build the Boost DLLs and LIBs by using these commands from Command Line (or PowerShell). Open CMD/PowerShell inside the Boost dir and type in:

  ```
  bootstrap.bat
  ```

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

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

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

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

- 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:

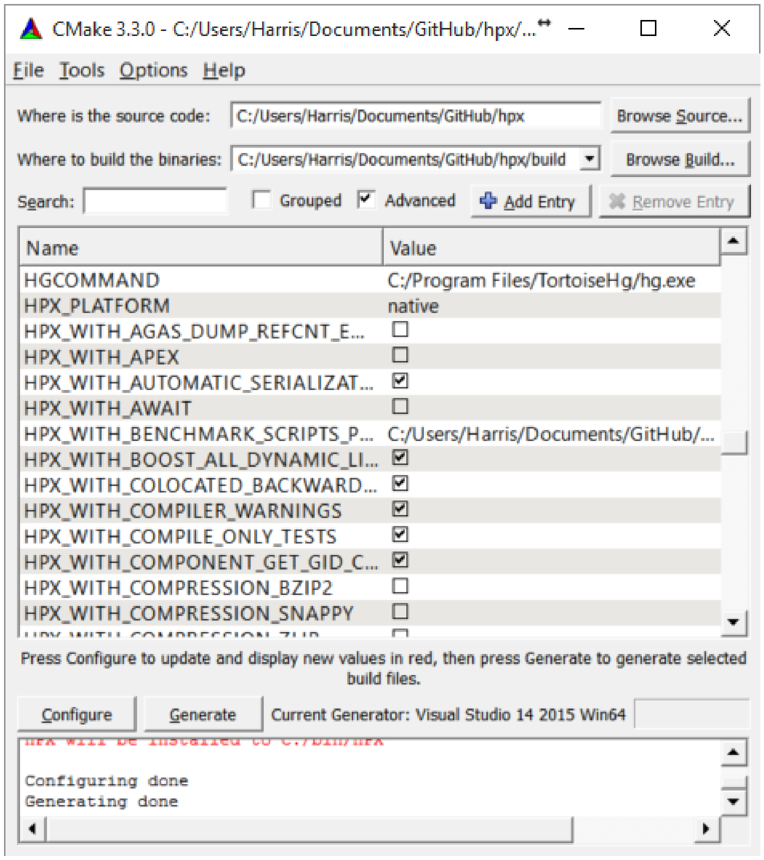

Fig. 2.3: Example CMake path settings.

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:

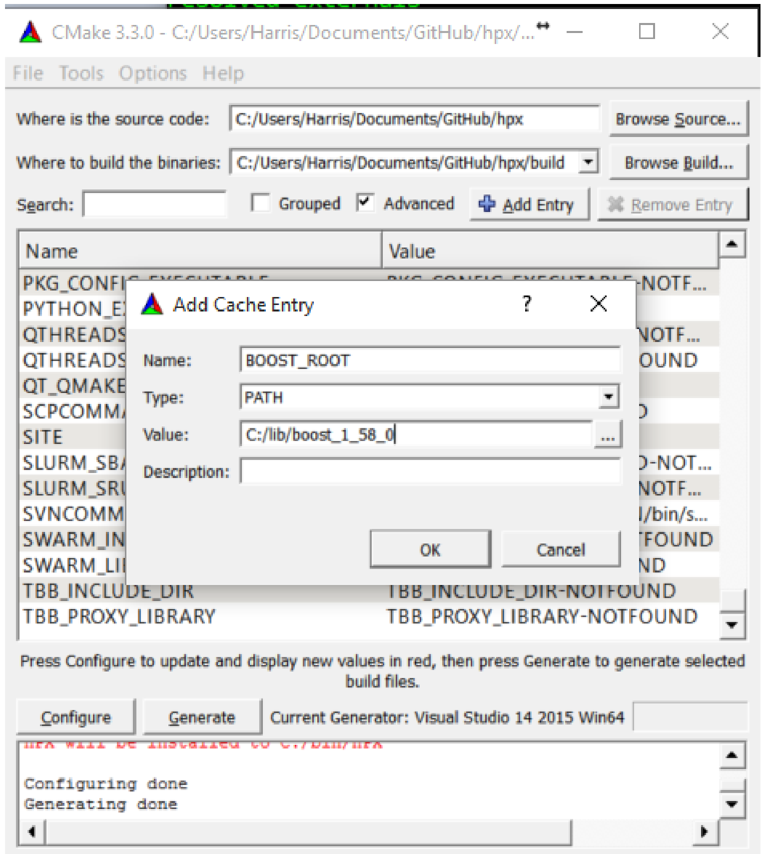

Fig. 2.4: Example CMake adding entry.

Alternatively, users could provide `BOOST_LIBRARYDIR` instead of `BOOST_ROOT`; the difference is that `BOOST_LIBRARYDIR` should point to the subdirectory inside Boost root where all the compiled DLLs/LIBs are. For example,

`BOOST_LIBRARYDIR` may point to the `bin.v2` subdirectory under the Boost rootdir. It is important to keep the meanings of these two variables separated from each other:

`BOOST_DIR` points to the ROOT folder of the Boost library. `BOOST_LIBRARYDIR` points to the subdir inside the Boost root folder where the compiled binaries are.

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

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

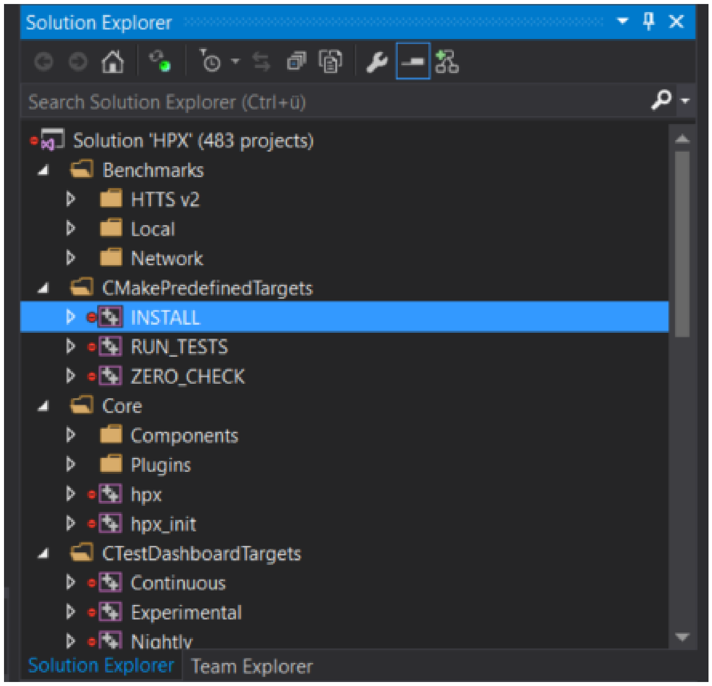

Fig. 2.5: Visual Studio INSTALL target.

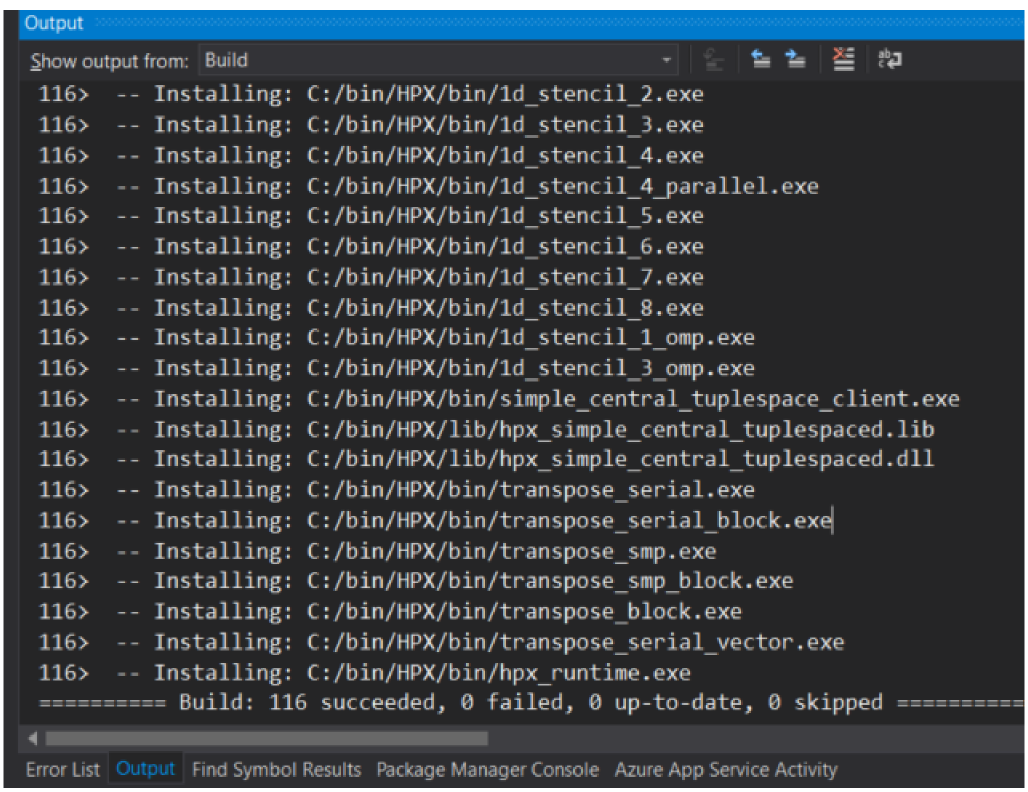

Fig. 2.6: Visual Studio build output.

How to install *HPX* on Fedora distributions

Important: There are official *HPX* packages for Fedora. Unless you want to customize your, build you may want to start off with the official packages. Instructions can be found on the *HPX* Downloads[91] page.

Note: This section of the manual is based off of our collaborator Patrick Diehl's blog post Installing |hpx| on Fedora 22[92].

- Install all packages for minimal installation:

```
sudo dnf install gcc-c++ cmake boost-build boost boost-devel hwloc-devel \
  hwloc gcc-gfortran papi-devel gperftools-devel docbook-dtds \
  docbook-style-xsl libsodium-devel doxygen boost-doc hdf5-devel \
  fop boost-devel boost-openmpi-devel boost-mpich-devel
```

- Get the development branch of *HPX*:

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

- Configure it with CMake:

```
cd hpx
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/opt/hpx ..
make -j
make install
```

Note: To build *HPX* without examples use:

```
cmake -DCMAKE_INSTALL_PREFIX=/opt/hpx -DHPX_WITH_EXAMPLES=Off ..
```

- Add the library path of *HPX* to ldconfig:

```
sudo echo /opt/hpx/lib > /etc/ld.so.conf.d/hpx.conf
sudo ldconfig
```

How to install *HPX* on Arch distributions

Important: There are *HPX* packages for Arch in the AUR. Unless you want to customize your build, you may want to start off with those. Instructions can be found on the *HPX* Downloads[93] page.

- Install all packages for a minimal installation:

[91] https://stellar-group.org/downloads/
[92] http://diehlpk.github.io/2015/08/04/hpx-fedora.html
[93] https://stellar-group.org/downloads/

```
sudo pacman -S gcc clang cmake boost hwloc gperftools
```

- For building the documentation, you will need to further install the following:

```
sudo pacman -S doxygen python-pip

pip install --user sphinx sphinx_rtd_theme breathe
```

The rest of the installation steps are the same as those for the Fedora or Unix variants.

How to install *HPX* on Debian-based distributions

- Install all packages for a minimal installation:

```
sudo apt install cmake libboost-all-dev hwloc libgoogle-perftools-dev
```

- To build the documentation you will need to further install the following:

```
sudo apt install doxygen python-pip

pip install --user sphinx sphinx_rtd_theme breathe
```

  or the following if you prefer to get Python packages from the Debian repositories:

```
sudo apt install doxygen python-sphinx python-sphinx-rtd-theme python-breathe
```

The rest of the installation steps are same as those for the Fedora or Unix variants.

CMake toolchains shipped with *HPX*

In order to compile *HPX* for various platforms, we provide a variety of toolchain files that take care of setting up various CMake variables like compilers, etc. They are located in the `cmake/toolchains` directory:

- *ARM-gcc*
- *BGION-gcc*
- *BGQ*
- *Cray*
- *CrayKNL*
- *CrayKNLStatic*
- *CrayStatic*
- *XeonPhi*

To use them, pass the `-DCMAKE_TOOLCHAIN_FILE=<toolchain>` argument to the CMake invocation.

ARM-gcc

```
# Copyright (c) 2015 Thomas Heller
#
# 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)
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_CROSSCOMPILING ON)
# Set the gcc Compiler
set(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++-4.8)
set(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc-4.8)
set(HPX_WITH_GENERIC_CONTEXT_COROUTINES
    ON
    CACHE BOOL "enable generic coroutines"
)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

BGION-gcc

```
# Copyright (c) 2014 John Biddiscombe
#
# 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)
# This is the default toolchain file to be used with CNK on a BlueGene/Q. It
# sets the appropriate compile flags and compiler such that HPX will compile.
# Note that you still need to provide Boost, hwloc and other utility libraries
# like a custom allocator yourself.
#
# Usage : cmake
# -DCMAKE_TOOLCHAIN_FILE=~/src/hpx/cmake/toolchains/BGION-gcc.cmake ~/src/hpx
#
set(CMAKE_SYSTEM_NAME Linux)
# Set the gcc Compiler
set(CMAKE_CXX_COMPILER g++)
set(CMAKE_C_COMPILER gcc)
# set(CMAKE_Fortran_COMPILER)
# Add flags we need for BGAS compilation
set(CMAKE_CXX_FLAGS_INIT
    "-D__powerpc__ -D__bgion__ -I/gpfs/bbp.cscs.ch/home/biddisco/src/bgas/rdmahelper "
    CACHE STRING "Initial compiler flags used to compile for BGAS"
)
# cmake-format: off
# the V1R2M2 includes are necessary for some hardware specific features
#    -DHPX_SMALL_STACK_SIZE=0x200000
#    -DHPX_MEDIUM_STACK_SIZE=0x200000
#    -DHPX_LARGE_STACK_SIZE=0x200000
#    -DHPX_HUGE_STACK_SIZE=0x200000
# cmake-format: on
set(CMAKE_EXE_LINKER_FLAGS_INIT
    "-L/gpfs/bbp.cscs.ch/apps/bgas/tools/gcc/gcc-4.8.2/install/lib64 -latomic -lrt"
```

(continues on next page)

(continued from previous page)

```
    CACHE STRING "BGAS flags"
)
set(CMAKE_C_FLAGS_INIT
    "-D__powerpc__ -I/gpfs/bbp.cscs.ch/home/biddisco/src/bgas/rdmahelper"
    CACHE STRING "BGAS flags"
)
# We do not perform cross compilation here ...
set(CMAKE_CROSSCOMPILING OFF)
# Set our platform name
set(HPX_PLATFORM "native")
# Disable generic coroutines (and use posix version)
set(HPX_WITH_GENERIC_CONTEXT_COROUTINES
    OFF
    CACHE BOOL "disable generic coroutines"
)
# BGAS nodes support ibverbs, but it is deprecated
set(HPX_WITH_PARCELPORT_VERBS
    OFF
    CACHE BOOL ""
)
# Always disable the tcp parcelport as it is non-functional on the BGQ.
set(HPX_WITH_PARCELPORT_TCP
    ON
    CACHE BOOL ""
)
# Always enable the tcp parcelport as it is currently the only way to
# communicate on the BGQ.
set(HPX_WITH_PARCELPORT_MPI
    ON
    CACHE BOOL ""
)
# We have a bunch of cores on the A2 processor ...
set(HPX_WITH_MAX_CPU_COUNT
    "64"
    CACHE STRING ""
)
# We have no custom malloc yet
if(NOT DEFINED HPX_WITH_MALLOC)
  set(HPX_WITH_MALLOC
      "system"
      CACHE STRING ""
  )
endif()
set(HPX_HIDDEN_VISIBILITY
    OFF
    CACHE BOOL ""
)
#
# Convenience setup for jb @ bbpbg2.cscs.ch
#
set(BOOST_ROOT "/gpfs/bbp.cscs.ch/home/biddisco/apps/gcc-4.8.2/boost_1_56_0")
set(HWLOC_ROOT "/gpfs/bbp.cscs.ch/home/biddisco/apps/gcc-4.8.2/hwloc-1.8.1")
set(CMAKE_BUILD_TYPE
    "Debug"
    CACHE STRING "Default build"
)
#
```

(continues on next page)

(continued from previous page)

```
# Testing flags
#
set(BUILD_TESTING
    ON
    CACHE BOOL "Testing enabled by default"
)
set(HPX_WITH_TESTS
    ON
    CACHE BOOL "Testing enabled by default"
)
set(HPX_WITH_TESTS_BENCHMARKS
    ON
    CACHE BOOL "Testing enabled by default"
)
set(HPX_WITH_TESTS_REGRESSIONS
    ON
    CACHE BOOL "Testing enabled by default"
)
set(HPX_WITH_TESTS_UNIT
    ON
    CACHE BOOL "Testing enabled by default"
)
set(HPX_WITH_TESTS_EXAMPLES
    ON
    CACHE BOOL "Testing enabled by default"
)
set(HPX_WITH_TESTS_EXTERNAL_BUILD
    OFF
    CACHE BOOL "Turn off build of cmake build tests"
)
set(DART_TESTING_TIMEOUT
    45
    CACHE STRING "Life is too short"
)
# HPX_WITH_STATIC_LINKING
```

BGQ

```
# Copyright (c) 2014 Thomas Heller
#
# 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)
#
# This is the default toolchain file to be used with CNK on a BlueGene/Q. It sets
# the appropriate compile flags and compiler such that HPX will compile.
# Note that you still need to provide Boost, hwloc and other utility libraries
# like a custom allocator yourself.
#
set(CMAKE_SYSTEM_NAME Linux)
# Set the Intel Compiler
set(CMAKE_CXX_COMPILER bgclang++11)
set(CMAKE_C_COMPILER bgclang)
# set(CMAKE_Fortran_COMPILER)
```

(continues on next page)

(continued from previous page)

```
set(MPI_CXX_COMPILER mpiclang++11)
set(MPI_C_COMPILER mpiclang)
# set(MPI_Fortran_COMPILER)
set(CMAKE_C_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_C_COMPILE_OBJECT
    "<CMAKE_C_COMPILER> -fPIC <DEFINES> <FLAGS> -o <OBJECT> -c <SOURCE>"
    CACHE STRING ""
)
set(CMAKE_C_LINK_EXECUTABLE
    "<CMAKE_C_COMPILER> -fPIC -dynamic <FLAGS> <CMAKE_C_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_C_CREATE_SHARED_LIBRARY
    "<CMAKE_C_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_
↪COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG>
↪<TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES> "
    CACHE STRING ""
)
set(CMAKE_CXX_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_CXX_COMPILE_OBJECT
    "<CMAKE_CXX_COMPILER> -fPIC <DEFINES> <FLAGS> -o <OBJECT> -c <SOURCE>"
    CACHE STRING ""
)
set(CMAKE_CXX_LINK_EXECUTABLE
    "<CMAKE_CXX_COMPILER> -fPIC -dynamic <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_CXX_CREATE_SHARED_LIBRARY
    "<CMAKE_CXX_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_
↪COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG>
↪<TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_Fortran_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_Fortran_COMPILE_OBJECT
    "<CMAKE_Fortran_COMPILER> -fPIC <DEFINES> <FLAGS> -o <OBJECT> -c <SOURCE>"
    CACHE STRING ""
)
set(CMAKE_Fortran_LINK_EXECUTABLE
    "<CMAKE_Fortran_COMPILER> -fPIC -dynamic <FLAGS> <CMAKE_Fortran_LINK_FLAGS> <LINK_
↪FLAGS> <OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
)
set(CMAKE_Fortran_CREATE_SHARED_LIBRARY
    "<CMAKE_Fortran_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_Fortran_FLAGS>
↪<LANGUAGE_COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_Fortran_FLAGS>
↪<SONAME_FLAG><TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES> "
```

(continues on next page)

(continued from previous page)

```
    CACHE STRING ""
)
# Disable searches in the default system paths. We are cross compiling after all
# and cmake might pick up wrong libraries that way
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
# We do a cross compilation here ...
set(CMAKE_CROSSCOMPILING ON)
# Set our platform name
set(HPX_PLATFORM "BlueGeneQ")
# Always disable the ibverbs parcelport as it is non-functional on the BGQ.
set(HPX_WITH_PARCELPORT_VERBS OFF)
# Always disable the tcp parcelport as it is non-functional on the BGQ.
set(HPX_WITH_PARCELPORT_TCP OFF)
# Always enable the mpi parcelport as it is currently the only way to
# communicate on the BGQ.
set(HPX_WITH_PARCELPORT_MPI ON)
# We have a bunch of cores on the BGQ ...
set(HPX_WITH_MAX_CPU_COUNT "64")
# We default to tbbmalloc as our allocator on the MIC
if(NOT DEFINED HPX_WITH_MALLOC)
  set(HPX_WITH_MALLOC
      "system"
      CACHE STRING ""
  )
endif()
```

Cray

```
# Copyright (c) 2014 Thomas Heller
#
# 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)
#
# This is the default toolchain file to be used with Intel Xeon PHIs. It sets
# the appropriate compile flags and compiler such that HPX will compile.
# Note that you still need to provide Boost, hwloc and other utility libraries
# like a custom allocator yourself.
#
# set(CMAKE_SYSTEM_NAME Cray-CNK-Intel)
if(HPX_WITH_STATIC_LINKING)
  set_property(GLOBAL PROPERTY TARGET_SUPPORTS_SHARED_LIBS FALSE)
else()
endif()
# Set the Cray Compiler Wrapper
set(CMAKE_CXX_COMPILER CC)
set(CMAKE_C_COMPILER cc)
set(CMAKE_Fortran_COMPILER ftn)
set(CMAKE_C_FLAGS_INIT
    ""
    CACHE STRING ""
```

(continues on next page)

(continued from previous page)

```
)
set(CMAKE_SHARED_LIBRARY_C_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_C_COMPILE_OBJECT
    "<CMAKE_C_COMPILER> -shared -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_C_LINK_EXECUTABLE
    "<CMAKE_C_COMPILER> -fPIC -dynamic <FLAGS> <CMAKE_C_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_C_CREATE_SHARED_LIBRARY
    "<CMAKE_C_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_
↪COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG>
↪<TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES> "
    CACHE STRING ""
)
set(CMAKE_CXX_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CXX_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_CXX_COMPILE_OBJECT
    "<CMAKE_CXX_COMPILER> -shared -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_CXX_LINK_EXECUTABLE
    "<CMAKE_CXX_COMPILER> -fPIC -dynamic <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_CXX_CREATE_SHARED_LIBRARY
    "<CMAKE_CXX_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_
↪COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG>
↪<TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>"
    CACHE STRING ""
)
```

(continues on next page)

(continued from previous page)

```
set(CMAKE_Fortran_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_Fortran_FLAGS
    "-fPIC"
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CREATE_Fortran_FLAGS
    "-shared"
    CACHE STRING ""
)
set(CMAKE_Fortran_COMPILE_OBJECT
    "<CMAKE_Fortran_COMPILER> -shared -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -
↪c <SOURCE>"
    CACHE STRING ""
)
set(CMAKE_Fortran_LINK_EXECUTABLE
    "<CMAKE_Fortran_COMPILER> -fPIC -dynamic <FLAGS> <CMAKE_Fortran_LINK_FLAGS> <LINK_
↪FLAGS> <OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
)
set(CMAKE_Fortran_CREATE_SHARED_LIBRARY
    "<CMAKE_Fortran_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_Fortran_FLAGS>
↪<LANGUAGE_COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_Fortran_FLAGS>
↪<SONAME_FLAG><TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES> "
    CACHE STRING ""
)
# Disable searches in the default system paths. We are cross compiling after all
# and cmake might pick up wrong libraries that way
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
set(HPX_WITH_PARCELPORT_TCP
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI_MULTITHREADED
    OFF
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_LIBFABRIC
    ON
    CACHE BOOL ""
)
set(HPX_PARCELPORT_LIBFABRIC_PROVIDER
    "gni"
    CACHE STRING "See libfabric docs for details, gni,verbs,psm2 etc etc"
)
set(HPX_PARCELPORT_LIBFABRIC_THROTTLE_SENDS
    "256"
    CACHE STRING "Max number of messages in flight at once"
)
```

(continues on next page)

(continued from previous page)

```
set(HPX_PARCELPORT_LIBFABRIC_WITH_DEV_MODE
    OFF
    CACHE BOOL "Custom libfabric logging flag"
)
set(HPX_PARCELPORT_LIBFABRIC_WITH_LOGGING
    OFF
    CACHE BOOL "Libfabric parcelport logging on/off flag"
)
set(HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD
    "4096"
    CACHE
      STRING
      "The threshold in bytes to when perform zero copy optimizations (default: 128)"
)
# We do a cross compilation here ...
set(CMAKE_CROSSCOMPILING
    ON
    CACHE BOOL ""
)
```

CrayKNL

```
# Copyright (c) 2014 Thomas Heller
#
# 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)
#
# This is the default toolchain file to be used with Intel Xeon PHIs. It sets
# the appropriate compile flags and compiler such that HPX will compile.
# Note that you still need to provide Boost, hwloc and other utility libraries
# like a custom allocator yourself.
#
if(HPX_WITH_STATIC_LINKING)
  set_property(GLOBAL PROPERTY TARGET_SUPPORTS_SHARED_LIBS FALSE)
else()
endif()
# Set the Cray Compiler Wrapper
set(CMAKE_CXX_COMPILER CC)
set(CMAKE_C_COMPILER cc)
set(CMAKE_Fortran_COMPILER ftn)
set(CMAKE_C_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_C_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_C_COMPILE_OBJECT
```

(continues on next page)

(continued from previous page)

```
    "<CMAKE_C_COMPILER> -shared -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_C_LINK_EXECUTABLE
    "<CMAKE_C_COMPILER> -fPIC <FLAGS> <CMAKE_C_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o
↪<TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_C_CREATE_SHARED_LIBRARY
    "<CMAKE_C_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_
↪COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG>
↪<TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES> "
    CACHE STRING ""
)
#
set(CMAKE_CXX_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CXX_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS
    "-fPIC -shared"
    CACHE STRING ""
)
set(CMAKE_CXX_COMPILE_OBJECT
    "<CMAKE_CXX_COMPILER> -shared -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_CXX_LINK_EXECUTABLE
    "<CMAKE_CXX_COMPILER> -fPIC -dynamic <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_CXX_CREATE_SHARED_LIBRARY
    "<CMAKE_CXX_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_
↪COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG>
↪<TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>"
    CACHE STRING ""
)
#
set(CMAKE_Fortran_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_SHARED_LIBRARY_Fortran_FLAGS
    "-fPIC"
    CACHE STRING ""
)
```

(continues on next page)

(continued from previous page)

```
set(CMAKE_SHARED_LIBRARY_CREATE_Fortran_FLAGS
    "-shared"
    CACHE STRING ""
)
set(CMAKE_Fortran_COMPILE_OBJECT
    "<CMAKE_Fortran_COMPILER> -shared -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -
↪c <SOURCE>"
    CACHE STRING ""
)
set(CMAKE_Fortran_LINK_EXECUTABLE
    "<CMAKE_Fortran_COMPILER> -fPIC <FLAGS> <CMAKE_Fortran_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
)
set(CMAKE_Fortran_CREATE_SHARED_LIBRARY
    "<CMAKE_Fortran_COMPILER> -fPIC -shared <CMAKE_SHARED_LIBRARY_Fortran_FLAGS>
↪<LANGUAGE_COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_Fortran_FLAGS>
↪<SONAME_FLAG><TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES> "
    CACHE STRING ""
)
#
# Disable searches in the default system paths. We are cross compiling after all
# and cmake might pick up wrong libraries that way
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
set(HPX_WITH_PARCELPORT_TCP
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI_MULTITHREADED
    OFF
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_LIBFABRIC
    ON
    CACHE BOOL ""
)
set(HPX_PARCELPORT_LIBFABRIC_PROVIDER
    "gni"
    CACHE STRING "See libfabric docs for details, gni,verbs,psm2 etc etc"
)
set(HPX_PARCELPORT_LIBFABRIC_THROTTLE_SENDS
    "256"
    CACHE STRING "Max number of messages in flight at once"
)
set(HPX_PARCELPORT_LIBFABRIC_WITH_DEV_MODE
    OFF
    CACHE BOOL "Custom libfabric logging flag"
)
set(HPX_PARCELPORT_LIBFABRIC_WITH_LOGGING
    OFF
    CACHE BOOL "Libfabric parcelport logging on/off flag"
```

(continues on next page)

(continued from previous page)

```
)
set(HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD
    "4096"
    CACHE
      STRING
      "The threshold in bytes to when perform zero copy optimizations (default: 128)"
)
# Set the TBBMALLOC_PLATFORM correctly so that find_package(TBBMalloc) sets the
# right hints
set(TBBMALLOC_PLATFORM
    "mic-knl"
    CACHE STRING ""
)
# We have a bunch of cores on the MIC ... increase the default
set(HPX_WITH_MAX_CPU_COUNT
    "512"
    CACHE STRING ""
)
# We do a cross compilation here ...
set(CMAKE_CROSSCOMPILING
    ON
    CACHE BOOL ""
)
# RDTSCP is available on Xeon/Phis
set(HPX_WITH_RDTSCP
    ON
    CACHE BOOL ""
)
```

CrayKNLStatic

```
# Copyright (c) 2014-2017 Thomas Heller
# Copyright (c) 2017      Bryce Adelstein Lelbach
#
# 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)
set(HPX_WITH_STATIC_LINKING
    ON
    CACHE BOOL ""
)
set(HPX_WITH_STATIC_EXE_LINKING
    ON
    CACHE BOOL ""
)
set_property(GLOBAL PROPERTY TARGET_SUPPORTS_SHARED_LIBS FALSE)
# Set the Cray Compiler Wrapper
set(CMAKE_CXX_COMPILER CC)
set(CMAKE_C_COMPILER cc)
set(CMAKE_Fortran_COMPILER ftn)
set(CMAKE_C_FLAGS_INIT
    ""
    CACHE STRING ""
)
```

(continues on next page)

(continued from previous page)

```
set(CMAKE_C_COMPILE_OBJECT
    "<CMAKE_C_COMPILER> -static -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_C_LINK_EXECUTABLE
    "<CMAKE_C_COMPILER> -fPIC <FLAGS> <CMAKE_C_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o
↪<TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_CXX_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_CXX_COMPILE_OBJECT
    "<CMAKE_CXX_COMPILER> -static -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_CXX_LINK_EXECUTABLE
    "<CMAKE_CXX_COMPILER> -fPIC <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS> <OBJECTS>␣
↪-o <TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_Fortran_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_Fortran_COMPILE_OBJECT
    "<CMAKE_Fortran_COMPILER> -static -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -
↪c <SOURCE>"
    CACHE STRING ""
)
set(CMAKE_Fortran_LINK_EXECUTABLE
    "<CMAKE_Fortran_COMPILER> -fPIC <FLAGS> <CMAKE_Fortran_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
)
# Disable searches in the default system paths. We are cross compiling after all
# and cmake might pick up wrong libraries that way
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
set(HPX_WITH_PARCELPORT_TCP
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI_MULTITHREADED
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_LIBFABRIC
    ON
```

(continues on next page)

(continued from previous page)

```
    CACHE BOOL ""
)
set(HPX_PARCELPORT_LIBFABRIC_PROVIDER
    "gni"
    CACHE STRING "See libfabric docs for details, gni,verbs,psm2 etc etc"
)
set(HPX_PARCELPORT_LIBFABRIC_THROTTLE_SENDS
    "256"
    CACHE STRING "Max number of messages in flight at once"
)
set(HPX_PARCELPORT_LIBFABRIC_WITH_DEV_MODE
    OFF
    CACHE BOOL "Custom libfabric logging flag"
)
set(HPX_PARCELPORT_LIBFABRIC_WITH_LOGGING
    OFF
    CACHE BOOL "Libfabric parcelport logging on/off flag"
)
set(HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD
    "4096"
    CACHE
      STRING
      "The threshold in bytes to when perform zero copy optimizations (default: 128)"
)
# Set the TBBMALLOC_PLATFORM correctly so that find_package(TBBMalloc) sets the
# right hints
set(TBBMALLOC_PLATFORM
    "mic-knl"
    CACHE STRING ""
)
# We have a bunch of cores on the MIC ... increase the default
set(HPX_WITH_MAX_CPU_COUNT
    "512"
    CACHE STRING ""
)
# We do a cross compilation here ...
set(CMAKE_CROSSCOMPILING
    ON
    CACHE BOOL ""
)
# RDTSCP is available on Xeon/Phis
set(HPX_WITH_RDTSCP
    ON
    CACHE BOOL ""
)
```

CrayStatic

```
# Copyright (c) 2014-2017 Thomas Heller
# Copyright (c) 2017      Bryce Adelstein Lelbach
#
# 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)
set(HPX_WITH_STATIC_LINKING
    ON
    CACHE BOOL ""
)
set(HPX_WITH_STATIC_EXE_LINKING
    ON
    CACHE BOOL ""
)
set_property(GLOBAL PROPERTY TARGET_SUPPORTS_SHARED_LIBS FALSE)
# Set the Cray Compiler Wrapper
set(CMAKE_CXX_COMPILER CC)
set(CMAKE_C_COMPILER cc)
set(CMAKE_Fortran_COMPILER ftn)
set(CMAKE_C_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_C_COMPILE_OBJECT
    "<CMAKE_C_COMPILER> -static -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_C_LINK_EXECUTABLE
    "<CMAKE_C_COMPILER> -fPIC <FLAGS> <CMAKE_C_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o
↪<TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_CXX_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_CXX_COMPILE_OBJECT
    "<CMAKE_CXX_COMPILER> -static -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c
↪<SOURCE>"
    CACHE STRING ""
)
set(CMAKE_CXX_LINK_EXECUTABLE
    "<CMAKE_CXX_COMPILER> -fPIC <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS> <OBJECTS>␣
↪-o <TARGET> <LINK_LIBRARIES>"
    CACHE STRING ""
)
set(CMAKE_Fortran_FLAGS_INIT
    ""
    CACHE STRING ""
)
set(CMAKE_Fortran_COMPILE_OBJECT
    "<CMAKE_Fortran_COMPILER> -static -fPIC <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -
↪c <SOURCE>"
    CACHE STRING ""
```

(continues on next page)

(continued from previous page)

```
)
set(CMAKE_Fortran_LINK_EXECUTABLE
    "<CMAKE_Fortran_COMPILER> -fPIC <FLAGS> <CMAKE_Fortran_LINK_FLAGS> <LINK_FLAGS>
↪<OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
)
# Disable searches in the default system paths. We are cross compiling after all
# and cmake might pick up wrong libraries that way
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
# We do a cross compilation here ...
set(CMAKE_CROSSCOMPILING
    ON
    CACHE BOOL ""
)
# RDTSCP is available on Xeon/Phis
set(HPX_WITH_RDTSCP
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_TCP
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_MPI_MULTITHREADED
    ON
    CACHE BOOL ""
)
set(HPX_WITH_PARCELPORT_LIBFABRIC
    ON
    CACHE BOOL ""
)
set(HPX_PARCELPORT_LIBFABRIC_PROVIDER
    "gni"
    CACHE STRING "See libfabric docs for details, gni,verbs,psm2 etc etc"
)
set(HPX_PARCELPORT_LIBFABRIC_THROTTLE_SENDS
    "256"
    CACHE STRING "Max number of messages in flight at once"
)
set(HPX_PARCELPORT_LIBFABRIC_WITH_DEV_MODE
    OFF
    CACHE BOOL "Custom libfabric logging flag"
)
set(HPX_PARCELPORT_LIBFABRIC_WITH_LOGGING
    OFF
    CACHE BOOL "Libfabric parcelport logging on/off flag"
)
set(HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD
    "4096"
    CACHE
      STRING
```

(continues on next page)

(continued from previous page)

```
    "The threshold in bytes to when perform zero copy optimizations (default: 128)"
)
```

XeonPhi

```
# Copyright (c) 2014 Thomas Heller
#
# 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)
#
# This is the default toolchain file to be used with Intel Xeon PHIs. It sets
# the appropriate compile flags and compiler such that HPX will compile.
# Note that you still need to provide Boost, hwloc and other utility libraries
# like a custom allocator yourself.
#
set(CMAKE_SYSTEM_NAME Linux)
# Set the Intel Compiler
set(CMAKE_CXX_COMPILER icpc)
set(CMAKE_C_COMPILER icc)
set(CMAKE_Fortran_COMPILER ifort)
# Add the -mmic compile flag such that everything will be compiled for the
# correct platform
set(CMAKE_CXX_FLAGS_INIT
    "-mmic"
    CACHE STRING "Initial compiler flags used to compile for the Xeon Phi"
)
set(CMAKE_C_FLAGS_INIT
    "-mmic"
    CACHE STRING "Initial compiler flags used to compile for the Xeon Phi"
)
set(CMAKE_Fortran_FLAGS_INIT
    "-mmic"
    CACHE STRING "Initial compiler flags used to compile for the Xeon Phi"
)
# Disable searches in the default system paths. We are cross compiling after all
# and cmake might pick up wrong libraries that way
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
# We do a cross compilation here ...
set(CMAKE_CROSSCOMPILING ON)
# Set our platform name
set(HPX_PLATFORM "XeonPhi")
# Always disable the ibverbs parcelport as it is non-functional on the BGQ.
set(HPX_WITH_PARCELPORT_VERBS
    OFF
    CACHE
      BOOL
      "Enable the ibverbs based parcelport. This is currently an experimental feature"
)
set(HPX_WITH_PARCELPORT_MPI
    ON
```

(continues on next page)

(continued from previous page)

```
    CACHE BOOL "Enable the MPI based parcelport."
)
# We have a bunch of cores on the MIC ... increase the default
set(HPX_WITH_MAX_CPU_COUNT
    "256"
    CACHE STRING ""
)
# We default to tbbmalloc as our allocator on the MIC
if(NOT DEFINED HPX_WITH_MALLOC)
  set(HPX_WITH_MALLOC
      "tbbmalloc"
      CACHE STRING ""
  )
endif()
# Set the TBBMALLOC_PLATFORM correctly so that find_package(TBBMalloc) sets the
# right hints
set(TBBMALLOC_PLATFORM
    "mic"
    CACHE STRING ""
)
set(HPX_HIDDEN_VISIBILITY
    OFF
    CACHE BOOL
          "Use -fvisibility=hidden for builds on platforms which support it"
)
# RDTSC is available on Xeon/Phis
set(HPX_WITH_RDTSC
    ON
    CACHE BOOL ""
)
```

CMake variables used to configure *HPX*

In order to configure *HPX*, you can set a variety of options to allow CMake to generate your specific makefiles/project files.

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_ACTION_BASE_COMPATIBILITY:BOOL*
- *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_COMPILER_WARNINGS:BOOL*
- *HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL*
- *HPX_WITH_COMPRESSION_BZIP2:BOOL*
- *HPX_WITH_COMPRESSION_SNAPPY:BOOL*
- *HPX_WITH_COMPRESSION_ZLIB:BOOL*
- *HPX_WITH_CUDA:BOOL*
- *HPX_WITH_CUDA_CLANG:BOOL*
- *HPX_WITH_CUDA_COMPUTE:BOOL*
- *HPX_WITH_DATAPAR_VC:BOOL*
- *HPX_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_WITH_DISABLED_SIGNAL_EXCEPTION_HANDLERS:BOOL*
- *HPX_WITH_DYNAMIC_HPX_MAIN:BOOL*
- *HPX_WITH_EMBEDDED_THREAD_POOLS_COMPATIBILITY:BOOL*
- *HPX_WITH_FAULT_TOLERANCE:BOOL*
- *HPX_WITH_FORTRAN:BOOL*
- *HPX_WITH_FULL_RPATH:BOOL*
- *HPX_WITH_GCC_VERSION_CHECK:BOOL*
- *HPX_WITH_GENERIC_CONTEXT_COROUTINES:BOOL*
- *HPX_WITH_HIDDEN_VISIBILITY:BOOL*
- *HPX_WITH_INIT_START_OVERLOADS_COMPATIBILITY:BOOL*
- *HPX_WITH_LOGGING:BOOL*
- *HPX_WITH_MALLOC:STRING*
- *HPX_WITH_NATIVE_TLS:BOOL*
- *HPX_WITH_NICE_THREADLEVEL:BOOL*
- *HPX_WITH_PARCEL_COALESCING:BOOL*
- *HPX_WITH_POOL_EXECUTOR_COMPATIBILITY:BOOL*
- *HPX_WITH_PROMISE_ALIAS_COMPATIBILITY:BOOL*
- *HPX_WITH_REGISTER_THREAD_COMPATIBILITY:BOOL*
- *HPX_WITH_REGISTER_THREAD_OVERLOADS_COMPATIBILITY:BOOL*
- *HPX_WITH_RUN_MAIN_EVERYWHERE:BOOL*

- *HPX_WITH_STACKOVERFLOW_DETECTION:BOOL*
- *HPX_WITH_STATIC_LINKING:BOOL*
- *HPX_WITH_THREAD_AWARE_TIMER_COMPATIBILITY:BOOL*
- *HPX_WITH_THREAD_EXECUTORS_COMPATIBILITY:BOOL*
- *HPX_WITH_THREAD_POOL_OS_EXECUTOR_COMPATIBILITY:BOOL*
- *HPX_WITH_UNITY_BUILD:BOOL*
- *HPX_WITH_UNSCOPED_ENUM_COMPATIBILITY:BOOL*
- *HPX_WITH_VIM_YCM:BOOL*
- *HPX_WITH_ZERO_COPY_SERIALIZATION_THRESHOLD:STRING*

HPX_WITH_ACTION_BASE_COMPATIBILITY:BOOL
Enable deprecated action bases (default: ON)

HPX_WITH_ASYNC_CUDA:BOOL
Enable HPX CUDA compute capability (parallel algorithms) module (default: OFF) - note: CUDA futures may be used without CUDA Compute

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

HPX_WITH_BENCHMARK_SCRIPTS_PATH:PATH
Directory to place batch scripts in

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

HPX_WITH_COMPILER_WARNINGS:BOOL
Enable compiler warnings (default: ON)

HPX_WITH_COMPILER_WARNINGS_AS_ERRORS:BOOL
Turn compiler warnings into errors (default: OFF)

HPX_WITH_COMPRESSION_BZIP2:BOOL
Enable bzip2 compression for parcel data (default: OFF).

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

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

HPX_WITH_CUDA:BOOL
Enable HPX_WITH_ASYNC_CUDA (CUDA futures) and HPX_WITH_CUDA_COMPUTE (CUDA enabled parallel algorithms) (default: OFF)

HPX_WITH_CUDA_CLANG:BOOL
Use clang to compile CUDA code (default: OFF)

HPX_WITH_CUDA_COMPUTE:BOOL
Enable HPX CUDA compute capability (parallel algorithms) module (default: OFF) - note: enabling this also enables CUDA futures via HPX_WITH_ASYNC_CUDA

HPX_WITH_DATAPAR_VC:BOOL
Enable data parallel algorithm support using the external Vc 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_EMBEDDED_THREAD_POOLS_COMPATIBILITY:BOOL
Enable deprecated embedded thread pools (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_FORTRAN:BOOL
Enable or disable the compilation of Fortran examples using HPX

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_INIT_START_OVERLOADS_COMPATIBILITY:BOOL
Enable deprecated init() and start() overloads functions (default: ON)

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

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

HPX_WITH_NATIVE_TLS:BOOL
Use native TLS support if available (default:)

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_POOL_EXECUTOR_COMPATIBILITY:BOOL
Enable deprecated pool executor (default: ON)

HPX_WITH_PROMISE_ALIAS_COMPATIBILITY:BOOL
Enable deprecated alias of hpx::promise to hpx::lcos::promise (default: ON)

HPX_WITH_REGISTER_THREAD_COMPATIBILITY:BOOL
Enable deprecated register_thread/work functions in the hpx::applier namespace (default: ON)

HPX_WITH_REGISTER_THREAD_OVERLOADS_COMPATIBILITY:BOOL
Enable deprecated register_thread/work overloads (default: ON)

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_THREAD_AWARE_TIMER_COMPATIBILITY:BOOL
Enable deprecated thread_aware_timer (default: ON)

HPX_WITH_THREAD_EXECUTORS_COMPATIBILITY:BOOL
Enable deprecated thread executors (default: ON)

HPX_WITH_THREAD_POOL_OS_EXECUTOR_COMPATIBILITY:BOOL
Enable deprecated thread pool executors (default: ON)

HPX_WITH_UNITY_BUILD:BOOL
Enable unity build for certain build targets (experimental, requires CMake 3.16 or newer) (default OFF)

HPX_WITH_UNSCOPED_ENUM_COMPATIBILITY:BOOL
Enable deprecated unscoped enums (default: ON)

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_COMPILE_ONLY_TESTS:BOOL*
- *HPX_WITH_DEFAULT_TARGETS: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_IO_COUNTERS:BOOL*
- *HPX_WITH_PSEUDO_DEPENDENCIES: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_COMPILE_ONLY_TESTS:BOOL
Create build system support for compile time only HPX tests (default ON)

HPX_WITH_DEFAULT_TARGETS:BOOL
Associate the core HPX library with the default build target (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_IO_COUNTERS:BOOL
Enable IO counters (default: ON)

HPX_WITH_PSEUDO_DEPENDENCIES:BOOL
Force creating pseudo targets and pseudo dependencies (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_SCHEDULER_MAX_TERMINATED_THREADS:STRING*
- *HPX_WITH_COROUTINE_COUNTERS:BOOL*
- *HPX_WITH_IO_POOL:BOOL*
- *HPX_WITH_MAX_CPU_COUNT:STRING*
- *HPX_WITH_MAX_NUMA_DOMAIN_COUNT:STRING*
- *HPX_WITH_MORE_THAN_64_THREADS:BOOL*
- *HPX_WITH_SCHEDULER_LOCAL_STORAGE:BOOL*
- *HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL*
- *HPX_WITH_SPINLOCK_POOL_NUM:STRING*
- *HPX_WITH_STACKTRACES:BOOL*
- *HPX_WITH_STACKTRACES_DEMANGLE_SYMBOLS:BOOL*
- *HPX_WITH_STACKTRACES_STATIC_SYMBOLS:BOOL*
- *HPX_WITH_SWAP_CONTEXT_EMULATION:BOOL*
- *HPX_WITH_THREAD_BACKTRACE_DEPTH:STRING*
- *HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION:BOOL*
- *HPX_WITH_THREAD_CREATION_AND_CLEANUP_RATES:BOOL*
- *HPX_WITH_THREAD_CUMULATIVE_COUNTS:BOOL*
- *HPX_WITH_THREAD_IDLE_RATES:BOOL*
- *HPX_WITH_THREAD_LOCAL_STORAGE:BOOL*
- *HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF:BOOL*
- *HPX_WITH_THREAD_QUEUE_WAITTIME:BOOL*
- *HPX_WITH_THREAD_SCHEDULERS:STRING*
- *HPX_WITH_THREAD_STACK_MMAP:BOOL*
- *HPX_WITH_THREAD_STEALING_COUNTS:BOOL*
- *HPX_WITH_THREAD_TARGET_ADDRESS:BOOL*

- *HPX_WITH_TIMER_POOL:BOOL*

HPX_SCHEDULER_MAX_TERMINATED_THREADS:STRING
[Deprecated] Maximum number of terminated threads collected before those are cleaned up (default: 100)

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_MORE_THAN_64_THREADS:BOOL
HPX applications will be able to run on more than 64 cores (This variable is deprecated. The value is derived from HPX_WITH_MAX_CPU_COUNT instead.)

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

HPX_WITH_SPINLOCK_DEADLOCK_DETECTION:BOOL
Enable spinlock deadlock detection (default: OFF)

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

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

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

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

HPX_WITH_SWAP_CONTEXT_EMULATION:BOOL
Emulate SwapContext API for coroutines (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_SCHEDULERS:STRING
Which thread schedulers are built. Options are: all, abp-priority, local, static-priority, static, shared-priority. For multiple enabled schedulers, separate with a semicolon (default: all)

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*
- *HPX_WITH_PARCELPORT_ACTION_COUNTERS:BOOL*
- *HPX_WITH_PARCELPORT_LIBFABRIC:BOOL*
- *HPX_WITH_PARCELPORT_MPI:BOOL*
- *HPX_WITH_PARCELPORT_TCP:BOOL*
- *HPX_WITH_PARCELPORT_VERBS:BOOL*
- *HPX_WITH_PARCEL_PROFILING:BOOL*

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

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

HPX_WITH_PARCELPORT_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_PARCELPORT_VERBS:BOOL
Enable the ibverbs based parcelport. This is currently an experimental feature

HPX_WITH_PARCEL_PROFILING:BOOL
Enable profiling data for parcels

Profiling options

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

HPX_WITH_APEX:BOOL
Enable APEX instrumentation support.

HPX_WITH_GOOGLE_PERFTOOLS:BOOL
Enable Google Perftools 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*
- *HPX_WITH_THREAD_DESCRIPTION_FULL:BOOL*
- *HPX_WITH_THREAD_GUARD_PAGE:BOOL*
- *HPX_WITH_VALGRIND:BOOL*
- *HPX_WITH_VERIFY_LOCKS:BOOL*
- *HPX_WITH_VERIFY_LOCKS_BACKTRACE:BOOL*
- *HPX_WITH_VERIFY_LOCKS_GLOBALLY:BOOL*

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

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

HPX_WITH_SANITIZERS:BOOL
Configure with sanitizer instrumentation support.

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

Modules options

- *HPX_ACTIONS_BASE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ACTIONS_BASE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ACTIONS_BASE_WITH_TESTS:BOOL*
- *HPX_ACTIONS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ACTIONS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ACTIONS_WITH_TESTS:BOOL*
- *HPX_AFFINITY_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_AFFINITY_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_AFFINITY_WITH_TESTS:BOOL*
- *HPX_ALGORITHMS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ALGORITHMS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ALGORITHMS_WITH_TESTS:BOOL*
- *HPX_ALLOCATOR_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ALLOCATOR_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ALLOCATOR_SUPPORT_WITH_TESTS:BOOL*
- *HPX_ASIO_WITH_COMPATIBILITY_HEADERS:BOOL*

- *HPX_ASIO_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ASIO_WITH_TESTS:BOOL*
- *HPX_ASSERTION_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ASSERTION_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ASSERTION_WITH_TESTS:BOOL*
- *HPX_ASYNC_BASE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ASYNC_BASE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ASYNC_BASE_WITH_TESTS:BOOL*
- *HPX_ASYNC_COMBINATORS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ASYNC_COMBINATORS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ASYNC_COMBINATORS_WITH_TESTS:BOOL*
- *HPX_ASYNC_DISTRIBUTED_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ASYNC_DISTRIBUTED_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ASYNC_DISTRIBUTED_WITH_TESTS:BOOL*
- *HPX_ASYNC_LOCAL_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ASYNC_LOCAL_WITH_TESTS:BOOL*
- *HPX_BATCH_ENVIRONMENTS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_BATCH_ENVIRONMENTS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_BATCH_ENVIRONMENTS_WITH_TESTS:BOOL*
- *HPX_CACHE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_CACHE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_CACHE_WITH_TESTS:BOOL*
- *HPX_CHECKPOINT_BASE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_CHECKPOINT_BASE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_CHECKPOINT_BASE_WITH_TESTS:BOOL*
- *HPX_CHECKPOINT_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_CHECKPOINT_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_CHECKPOINT_WITH_TESTS:BOOL*
- *HPX_COLLECTIVES_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_COLLECTIVES_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_COLLECTIVES_WITH_TESTS:BOOL*
- *HPX_COMMAND_LINE_HANDLING_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_COMMAND_LINE_HANDLING_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_COMMAND_LINE_HANDLING_WITH_TESTS:BOOL*
- *HPX_COMPONENTS_BASE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_COMPONENTS_BASE_WITH_DEPRECATION_WARNINGS:BOOL*

- *HPX_COMPONENTS_BASE_WITH_TESTS:BOOL*
- *HPX_COMPUTE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_COMPUTE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_COMPUTE_WITH_TESTS:BOOL*
- *HPX_CONCEPTS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_CONCEPTS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_CONCEPTS_WITH_TESTS:BOOL*
- *HPX_CONCURRENCY_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_CONCURRENCY_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_CONCURRENCY_WITH_TESTS:BOOL*
- *HPX_CONFIG_REGISTRY_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_CONFIG_REGISTRY_WITH_TESTS:BOOL*
- *HPX_CONFIG_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_CONFIG_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_CONFIG_WITH_TESTS:BOOL*
- *HPX_COROUTINES_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_COROUTINES_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_COROUTINES_WITH_TESTS:BOOL*
- *HPX_DATASTRUCTURES_WITH_ADAPT_STD_TUPLE:BOOL*
- *HPX_DATASTRUCTURES_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_DATASTRUCTURES_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_DATASTRUCTURES_WITH_TESTS:BOOL*
- *HPX_DEBUGGING_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_DEBUGGING_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_DEBUGGING_WITH_TESTS:BOOL*
- *HPX_ERRORS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ERRORS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ERRORS_WITH_TESTS:BOOL*
- *HPX_EXECUTION_BASE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_EXECUTION_BASE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_EXECUTION_BASE_WITH_TESTS:BOOL*
- *HPX_EXECUTION_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_EXECUTION_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_EXECUTION_WITH_TESTS:BOOL*
- *HPX_EXECUTORS_DISTRIBUTED_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_EXECUTORS_DISTRIBUTED_WITH_DEPRECATION_WARNINGS:BOOL*

- *HPX_EXECUTORS_DISTRIBUTED_WITH_TESTS:BOOL*
- *HPX_EXECUTORS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_EXECUTORS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_EXECUTORS_WITH_TESTS:BOOL*
- *HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY:BOOL*
- *HPX_FILESYSTEM_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_FILESYSTEM_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_FILESYSTEM_WITH_TESTS:BOOL*
- *HPX_FORMAT_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_FORMAT_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_FORMAT_WITH_TESTS:BOOL*
- *HPX_FUNCTIONAL_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_FUNCTIONAL_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_FUNCTIONAL_WITH_TESTS:BOOL*
- *HPX_FUTURES_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_FUTURES_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_FUTURES_WITH_TESTS:BOOL*
- *HPX_HARDWARE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_HARDWARE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_HARDWARE_WITH_TESTS:BOOL*
- *HPX_HASHING_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_HASHING_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_HASHING_WITH_TESTS:BOOL*
- *HPX_INCLUDE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_INCLUDE_WITH_TESTS:BOOL*
- *HPX_INIT_RUNTIME_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_INIT_RUNTIME_WITH_TESTS:BOOL*
- *HPX_IO_SERVICE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_IO_SERVICE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_IO_SERVICE_WITH_TESTS:BOOL*
- *HPX_ITERATOR_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ITERATOR_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ITERATOR_SUPPORT_WITH_TESTS:BOOL*
- *HPX_ITT_NOTIFY_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_ITT_NOTIFY_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_ITT_NOTIFY_WITH_TESTS:BOOL*

- *HPX_LCOS_DISTRIBUTED_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_LCOS_DISTRIBUTED_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_LCOS_DISTRIBUTED_WITH_TESTS:BOOL*
- *HPX_LCOS_LOCAL_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_LCOS_LOCAL_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_LCOS_LOCAL_WITH_TESTS:BOOL*
- *HPX_LOGGING_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_LOGGING_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_LOGGING_WITH_TESTS:BOOL*
- *HPX_MEMORY_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_MEMORY_WITH_TESTS:BOOL*
- *HPX_NAMING_BASE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_NAMING_BASE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_NAMING_BASE_WITH_TESTS:BOOL*
- *HPX_PACK_TRAVERSAL_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_PACK_TRAVERSAL_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_PACK_TRAVERSAL_WITH_TESTS:BOOL*
- *HPX_PERFORMANCE_COUNTERS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_PERFORMANCE_COUNTERS_WITH_TESTS:BOOL*
- *HPX_PLUGIN_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_PLUGIN_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_PLUGIN_WITH_TESTS:BOOL*
- *HPX_PREFIX_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_PREFIX_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_PREFIX_WITH_TESTS:BOOL*
- *HPX_PREPROCESSOR_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_PREPROCESSOR_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_PREPROCESSOR_WITH_TESTS:BOOL*
- *HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY:BOOL*
- *HPX_PROGRAM_OPTIONS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_PROGRAM_OPTIONS_WITH_TESTS:BOOL*
- *HPX_RESILIENCY_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_RESILIENCY_WITH_TESTS:BOOL*
- *HPX_RESOURCE_PARTITIONER_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_RESOURCE_PARTITIONER_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_RESOURCE_PARTITIONER_WITH_TESTS:BOOL*

- *HPX_RUNTIME_CONFIGURATION_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_RUNTIME_CONFIGURATION_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_RUNTIME_CONFIGURATION_WITH_TESTS:BOOL*
- *HPX_RUNTIME_LOCAL_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_RUNTIME_LOCAL_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_RUNTIME_LOCAL_WITH_TESTS:BOOL*
- *HPX_SCHEDULERS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_SCHEDULERS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_SCHEDULERS_WITH_TESTS:BOOL*
- *HPX_SEGMENTED_ALGORITHMS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_SEGMENTED_ALGORITHMS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_SEGMENTED_ALGORITHMS_WITH_TESTS:BOOL*
- *HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL*
- *HPX_SERIALIZATION_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_SERIALIZATION_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_SERIALIZATION_WITH_TESTS:BOOL*
- *HPX_STATIC_REINIT_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_STATIC_REINIT_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_STATIC_REINIT_WITH_TESTS:BOOL*
- *HPX_STATISTICS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_STATISTICS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_STATISTICS_WITH_TESTS:BOOL*
- *HPX_STRING_UTIL_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_STRING_UTIL_WITH_TESTS:BOOL*
- *HPX_SYNCHRONIZATION_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_SYNCHRONIZATION_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_SYNCHRONIZATION_WITH_TESTS:BOOL*
- *HPX_TESTING_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_TESTING_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_TESTING_WITH_TESTS:BOOL*
- *HPX_THREADING_BASE_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_THREADING_BASE_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_THREADING_BASE_WITH_TESTS:BOOL*
- *HPX_THREADING_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_THREADING_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_THREADING_WITH_TESTS:BOOL*

- *HPX_THREADMANAGER_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_THREADMANAGER_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_THREADMANAGER_WITH_TESTS:BOOL*
- *HPX_THREAD_EXECUTORS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_THREAD_EXECUTORS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_THREAD_EXECUTORS_WITH_TESTS:BOOL*
- *HPX_THREAD_POOLS_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_THREAD_POOLS_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_THREAD_POOLS_WITH_TESTS:BOOL*
- *HPX_THREAD_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_THREAD_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_THREAD_SUPPORT_WITH_TESTS:BOOL*
- *HPX_TIMED_EXECUTION_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_TIMED_EXECUTION_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_TIMED_EXECUTION_WITH_TESTS:BOOL*
- *HPX_TIMING_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_TIMING_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_TIMING_WITH_TESTS:BOOL*
- *HPX_TOPOLOGY_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_TOPOLOGY_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_TOPOLOGY_WITH_TESTS:BOOL*
- *HPX_TYPE_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_TYPE_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_TYPE_SUPPORT_WITH_TESTS:BOOL*
- *HPX_UTIL_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_UTIL_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_UTIL_WITH_TESTS:BOOL*
- *HPX_VERSION_WITH_COMPATIBILITY_HEADERS:BOOL*
- *HPX_VERSION_WITH_DEPRECATION_WARNINGS:BOOL*
- *HPX_VERSION_WITH_TESTS:BOOL*

HPX_ACTIONS_BASE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ACTIONS_BASE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ACTIONS_BASE_WITH_TESTS:BOOL
Build HPX actions_base module tests. (default: ON)

HPX_ACTIONS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ACTIONS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ACTIONS_WITH_TESTS:BOOL
Build HPX actions module tests. (default: ON)

HPX_AFFINITY_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_AFFINITY_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_AFFINITY_WITH_TESTS:BOOL
Build HPX affinity module tests. (default: ON)

HPX_ALGORITHMS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ALGORITHMS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ALGORITHMS_WITH_TESTS:BOOL
Build HPX algorithms module tests. (default: ON)

HPX_ALLOCATOR_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ALLOCATOR_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ALLOCATOR_SUPPORT_WITH_TESTS:BOOL
Build HPX allocator_support module tests. (default: ON)

HPX_ASIO_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ASIO_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ASIO_WITH_TESTS:BOOL
Build HPX asio module tests. (default: ON)

HPX_ASSERTION_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ASSERTION_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ASSERTION_WITH_TESTS:BOOL
Build HPX assertion module tests. (default: ON)

HPX_ASYNC_BASE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ASYNC_BASE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ASYNC_BASE_WITH_TESTS:BOOL
Build HPX async_base module tests. (default: ON)

HPX_ASYNC_COMBINATORS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ASYNC_COMBINATORS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ASYNC_COMBINATORS_WITH_TESTS:BOOL
Build HPX async_combinators module tests. (default: ON)

HPX_ASYNC_DISTRIBUTED_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ASYNC_DISTRIBUTED_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ASYNC_DISTRIBUTED_WITH_TESTS:BOOL
Build HPX async_distributed module tests. (default: ON)

HPX_ASYNC_LOCAL_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ASYNC_LOCAL_WITH_TESTS:BOOL
Build HPX async_local module tests. (default: ON)

HPX_BATCH_ENVIRONMENTS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_BATCH_ENVIRONMENTS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_BATCH_ENVIRONMENTS_WITH_TESTS:BOOL
Build HPX batch_environments module tests. (default: ON)

HPX_CACHE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_CACHE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_CACHE_WITH_TESTS:BOOL
Build HPX cache module tests. (default: ON)

HPX_CHECKPOINT_BASE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: OFF)

HPX_CHECKPOINT_BASE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_CHECKPOINT_BASE_WITH_TESTS:BOOL
Build HPX checkpoint_base module tests. (default: ON)

HPX_CHECKPOINT_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_CHECKPOINT_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_CHECKPOINT_WITH_TESTS:BOOL
Build HPX checkpoint module tests. (default: ON)

HPX_COLLECTIVES_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_COLLECTIVES_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_COLLECTIVES_WITH_TESTS:BOOL
Build HPX collectives module tests. (default: ON)

HPX_COMMAND_LINE_HANDLING_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_COMMAND_LINE_HANDLING_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_COMMAND_LINE_HANDLING_WITH_TESTS:BOOL
Build HPX command_line_handling module tests. (default: ON)

HPX_COMPONENTS_BASE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_COMPONENTS_BASE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_COMPONENTS_BASE_WITH_TESTS:BOOL
Build HPX components_base module tests. (default: ON)

HPX_COMPUTE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_COMPUTE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_COMPUTE_WITH_TESTS:BOOL
Build HPX compute module tests. (default: ON)

HPX_CONCEPTS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_CONCEPTS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_CONCEPTS_WITH_TESTS:BOOL
Build HPX concepts module tests. (default: ON)

HPX_CONCURRENCY_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_CONCURRENCY_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_CONCURRENCY_WITH_TESTS:BOOL
Build HPX concurrency module tests. (default: ON)

HPX_CONFIG_REGISTRY_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_CONFIG_REGISTRY_WITH_TESTS:BOOL
Build HPX config_registry module tests. (default: ON)

HPX_CONFIG_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: OFF)

HPX_CONFIG_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_CONFIG_WITH_TESTS:BOOL
Build HPX config module tests. (default: ON)

HPX_COROUTINES_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_COROUTINES_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_COROUTINES_WITH_TESTS:BOOL
Build HPX coroutines module tests. (default: ON)

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

HPX_DATASTRUCTURES_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_DATASTRUCTURES_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_DATASTRUCTURES_WITH_TESTS:BOOL
Build HPX datastructures module tests. (default: ON)

HPX_DEBUGGING_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_DEBUGGING_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_DEBUGGING_WITH_TESTS:BOOL
Build HPX debugging module tests. (default: ON)

HPX_ERRORS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ERRORS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ERRORS_WITH_TESTS:BOOL
Build HPX errors module tests. (default: ON)

HPX_EXECUTION_BASE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_EXECUTION_BASE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_EXECUTION_BASE_WITH_TESTS:BOOL
Build HPX execution_base module tests. (default: ON)

HPX_EXECUTION_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_EXECUTION_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_EXECUTION_WITH_TESTS:BOOL
Build HPX execution module tests. (default: ON)

HPX_EXECUTORS_DISTRIBUTED_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_EXECUTORS_DISTRIBUTED_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_EXECUTORS_DISTRIBUTED_WITH_TESTS:BOOL
Build HPX executors_distributed module tests. (default: ON)

HPX_EXECUTORS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_EXECUTORS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_EXECUTORS_WITH_TESTS:BOOL
Build HPX executors module tests. (default: ON)

HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY:BOOL
Enable Boost.FileSystem compatibility. (default: ON)

HPX_FILESYSTEM_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_FILESYSTEM_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_FILESYSTEM_WITH_TESTS:BOOL
Build HPX filesystem module tests. (default: ON)

HPX_FORMAT_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_FORMAT_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_FORMAT_WITH_TESTS:BOOL
Build HPX format module tests. (default: ON)

HPX_FUNCTIONAL_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_FUNCTIONAL_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_FUNCTIONAL_WITH_TESTS:BOOL
Build HPX functional module tests. (default: ON)

HPX_FUTURES_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_FUTURES_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_FUTURES_WITH_TESTS:BOOL
Build HPX futures module tests. (default: ON)

HPX_HARDWARE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_HARDWARE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_HARDWARE_WITH_TESTS:BOOL
Build HPX hardware module tests. (default: ON)

HPX_HASHING_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_HASHING_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_HASHING_WITH_TESTS:BOOL
Build HPX hashing module tests. (default: ON)

HPX_INCLUDE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_INCLUDE_WITH_TESTS:BOOL
Build HPX include module tests. (default: ON)

HPX_INIT_RUNTIME_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_INIT_RUNTIME_WITH_TESTS:BOOL
Build HPX init_runtime module tests. (default: ON)

HPX_IO_SERVICE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_IO_SERVICE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_IO_SERVICE_WITH_TESTS:BOOL
Build HPX io_service module tests. (default: ON)

HPX_ITERATOR_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ITERATOR_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ITERATOR_SUPPORT_WITH_TESTS:BOOL
Build HPX iterator_support module tests. (default: ON)

HPX_ITT_NOTIFY_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_ITT_NOTIFY_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_ITT_NOTIFY_WITH_TESTS:BOOL
Build HPX itt_notify module tests. (default: ON)

HPX_LCOS_DISTRIBUTED_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_LCOS_DISTRIBUTED_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_LCOS_DISTRIBUTED_WITH_TESTS:BOOL
Build HPX lcos_distributed module tests. (default: ON)

HPX_LCOS_LOCAL_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_LCOS_LOCAL_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_LCOS_LOCAL_WITH_TESTS:BOOL
Build HPX lcos_local module tests. (default: ON)

HPX_LOGGING_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_LOGGING_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_LOGGING_WITH_TESTS:BOOL
Build HPX logging module tests. (default: ON)

HPX_MEMORY_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_MEMORY_WITH_TESTS:BOOL
Build HPX memory module tests. (default: ON)

HPX_NAMING_BASE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: OFF)

HPX_NAMING_BASE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_NAMING_BASE_WITH_TESTS:BOOL
Build HPX naming_base module tests. (default: ON)

HPX_PACK_TRAVERSAL_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_PACK_TRAVERSAL_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_PACK_TRAVERSAL_WITH_TESTS:BOOL
Build HPX pack_traversal module tests. (default: ON)

HPX_PERFORMANCE_COUNTERS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_PERFORMANCE_COUNTERS_WITH_TESTS:BOOL
Build HPX performance_counters module tests. (default: ON)

HPX_PLUGIN_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_PLUGIN_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_PLUGIN_WITH_TESTS:BOOL
Build HPX plugin module tests. (default: ON)

HPX_PREFIX_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_PREFIX_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_PREFIX_WITH_TESTS:BOOL
Build HPX prefix module tests. (default: ON)

HPX_PREPROCESSOR_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_PREPROCESSOR_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_PREPROCESSOR_WITH_TESTS:BOOL
Build HPX preprocessor module tests. (default: ON)

HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY:BOOL
Enable Boost.ProgramOptions compatibility. (default: ON)

HPX_PROGRAM_OPTIONS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_PROGRAM_OPTIONS_WITH_TESTS:BOOL
Build HPX program_options module tests. (default: ON)

HPX_RESILIENCY_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_RESILIENCY_WITH_TESTS:BOOL
Build HPX resiliency module tests. (default: ON)

HPX_RESOURCE_PARTITIONER_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_RESOURCE_PARTITIONER_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_RESOURCE_PARTITIONER_WITH_TESTS:BOOL
Build HPX resource_partitioner module tests. (default: ON)

HPX_RUNTIME_CONFIGURATION_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_RUNTIME_CONFIGURATION_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_RUNTIME_CONFIGURATION_WITH_TESTS:BOOL
Build HPX runtime_configuration module tests. (default: ON)

HPX_RUNTIME_LOCAL_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_RUNTIME_LOCAL_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_RUNTIME_LOCAL_WITH_TESTS:BOOL
Build HPX runtime_local module tests. (default: ON)

HPX_SCHEDULERS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_SCHEDULERS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_SCHEDULERS_WITH_TESTS:BOOL
Build HPX schedulers module tests. (default: ON)

HPX_SEGMENTED_ALGORITHMS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_SEGMENTED_ALGORITHMS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_SEGMENTED_ALGORITHMS_WITH_TESTS:BOOL
Build HPX segmented_algorithms module tests. (default: ON)

HPX_SERIALIZATION_WITH_BOOST_TYPES:BOOL
Enable serialization of certain Boost types. (default: ON)

HPX_SERIALIZATION_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_SERIALIZATION_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_SERIALIZATION_WITH_TESTS:BOOL
Build HPX serialization module tests. (default: ON)

HPX_STATIC_REINIT_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_STATIC_REINIT_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_STATIC_REINIT_WITH_TESTS:BOOL
Build HPX static_reinit module tests. (default: ON)

HPX_STATISTICS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_STATISTICS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_STATISTICS_WITH_TESTS:BOOL
Build HPX statistics module tests. (default: ON)

HPX_STRING_UTIL_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_STRING_UTIL_WITH_TESTS:BOOL
Build HPX string_util module tests. (default: ON)

HPX_SYNCHRONIZATION_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_SYNCHRONIZATION_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_SYNCHRONIZATION_WITH_TESTS:BOOL
Build HPX synchronization module tests. (default: ON)

HPX_TESTING_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_TESTING_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_TESTING_WITH_TESTS:BOOL
Build HPX testing module tests. (default: ON)

HPX_THREADING_BASE_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_THREADING_BASE_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_THREADING_BASE_WITH_TESTS:BOOL
Build HPX threading_base module tests. (default: ON)

HPX_THREADING_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_THREADING_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_THREADING_WITH_TESTS:BOOL
Build HPX threading module tests. (default: ON)

HPX_THREADMANAGER_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_THREADMANAGER_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_THREADMANAGER_WITH_TESTS:BOOL
Build HPX threadmanager module tests. (default: ON)

HPX_THREAD_EXECUTORS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_THREAD_EXECUTORS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_THREAD_EXECUTORS_WITH_TESTS:BOOL
Build HPX thread_executors module tests. (default: ON)

HPX_THREAD_POOLS_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_THREAD_POOLS_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_THREAD_POOLS_WITH_TESTS:BOOL
Build HPX thread_pools module tests. (default: ON)

HPX_THREAD_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_THREAD_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_THREAD_SUPPORT_WITH_TESTS:BOOL
Build HPX thread_support module tests. (default: ON)

HPX_TIMED_EXECUTION_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_TIMED_EXECUTION_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_TIMED_EXECUTION_WITH_TESTS:BOOL
Build HPX timed_execution module tests. (default: ON)

HPX_TIMING_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_TIMING_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_TIMING_WITH_TESTS:BOOL
Build HPX timing module tests. (default: ON)

HPX_TOPOLOGY_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_TOPOLOGY_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_TOPOLOGY_WITH_TESTS:BOOL
Build HPX topology module tests. (default: ON)

HPX_TYPE_SUPPORT_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_TYPE_SUPPORT_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_TYPE_SUPPORT_WITH_TESTS:BOOL
Build HPX type_support module tests. (default: ON)

HPX_UTIL_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: ON)

HPX_UTIL_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_UTIL_WITH_TESTS:BOOL
Build HPX util module tests. (default: ON)

HPX_VERSION_WITH_COMPATIBILITY_HEADERS:BOOL
Enable compatibility headers for old headers. (default: OFF)

HPX_VERSION_WITH_DEPRECATION_WARNINGS:BOOL
Enable warnings for deprecated facilities. (default: On)

HPX_VERSION_WITH_TESTS:BOOL
Build HPX version module tests. (default: ON)

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.

2.5.3 Creating *HPX* projects

Using HPX with pkg-config

How to build *HPX* applications with pkg-config

After you are done installing *HPX*, you should be able to build the following program. It prints `Hello World!` on the *locality* you run it on.

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

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

Important: If the *HPX* libraries are in a path that is not found by the dynamic linker, you will need to add the path `$HPX_LOCATION/lib` to your linker search path (for example `LD_LIBRARY_PATH` on Linux).

To test the program, type:

```
./hello_world
```

which should print `Hello World!` and exit.

How to build *HPX* components with pkg-config

Let's try a more complex example involving an *HPX* component. An *HPX* component is a class that exposes *HPX* actions. *HPX* components are compiled into dynamically loaded modules called component libraries. Here's the source code:

hello_world_component.cpp

```
#include "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);
```

hello_world_component.hpp

```
#pragma once

#include <hpx/hpx.hpp>
#include <hpx/include/actions.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/include/components.hpp>
#include <hpx/serialization.hpp>

```

(continues on next page)

(continued from previous page)

```
#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 "hello_world_component.hpp"
#include <hpx/hpx_main.hpp>

int main(int argc, char* argv[])
{
    {
        // 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;
}
```

Copy the three source files above into three files (called `hello_world_component.cpp`, `hello_world_component.hpp` and `hello_world_client.cpp`, respectively).

Now, in the directory where you put the files, run the following command to build the component library. (where `$HPX_LOCATION` is the build directory or `CMAKE_INSTALL_PREFIX` you used while building *HPX*):

```
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig
c++ -o libhpx_hello_world.so hello_world_component.cpp \
   `pkg-config --cflags --libs hpx_component` \
    -lhpx_iostreams -DHPX_COMPONENT_NAME=hpx_hello_world
```

Now pick a directory in which to install your *HPX* component libraries. For this example, we'll choose a directory named `my_hpx_libs`:

```
mkdir ~/my_hpx_libs
mv libhpx_hello_world.so ~/my_hpx_libs
```

Note: *HPX* libraries have different names in debug and release mode. If you want to link against a debug *HPX* library, you need to use the `_debug` suffix for the pkg-config name. That means instead of `hpx_application` or `hpx_component` you will have to use `hpx_application_debug` or `hpx_component_debug`. Moreover, all referenced *HPX* components need to have a appended `d` suffix, e.g. instead of `-lhpx_iostreams` you will need to specify `-lhpx_iostreamsd`.

Important: If the *HPX* libraries are in a path that is not found by the dynamic linker. You need to add the path `$HPX_LOCATION/lib` to your linker search path (for example `LD_LIBRARY_PATH` on Linux).

Now, to build the application that uses this component (`hello_world_client.cpp`), we do:

```
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HPX_LOCATION/lib/pkgconfig
c++ -o hello_world_client hello_world_client.cpp \
   ``pkg-config --cflags --libs hpx_application``\
    -L${HOME}/my_hpx_libs -lhpx_hello_world -lhpx_iostreams
```

Important: When using pkg-config with *HPX*, the pkg-config flags must go after the `-o` flag.

Finally, you'll need to set your LD_LIBRARY_PATH before you can run the program. To run the program, type:

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$HOME/my_hpx_libs"
./hello_world_client
```

which should print `Hello HPX World!` and exit.

Using *HPX* with CMake-based projects

In addition to the pkg-config support discussed on the previous pages, *HPX* comes with full CMake support. In order to integrate *HPX* into existing or new CMakeLists.txt, you can leverage the find_package[94] command integrated into CMake. Following, is a Hello World component example using CMake.

Let's revisit what we have. We have three files that compose our example application:

- `hello_world_component.hpp`
- `hello_world_component.cpp`
- `hello_world_client.hpp`

The basic structure to include *HPX* into your CMakeLists.txt is shown here:

```
# Require a recent version of cmake
cmake_minimum_required(VERSION 3.13 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.

[94] https://www.cmake.org/cmake/help/latest/command/find_package.html

Using CMake targets

The recommended way of setting up your targets to use *HPX* is to link to the `HPX::hpx` CMake target:

```
target_link_libraries(hello_world_component PUBLIC HPX::hpx)
```

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

```
add_library(hello_world_component SHARED hello_world_component.cpp)
target_include_directories(hello_world_component PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

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

Creating a component requires setting two additional compile definitions:

```
target_compile_options(hello_world_component
  HPX_COMPONENT_NAME=hello_world
  HPX_COMPONENT_EXPORTS)
```

Instead of setting these definitions manually you may link to the `HPX::component` target, which sets `HPX_COMPONENT_NAME` to `hpx_<target_name>`, where `<target_name>` is the target name of your library. Note that these definitions should be `PRIVATE` to make sure these definitions are not propagated transitively to dependent targets.

In addition to making your library a component you can make it a plugin. To do so link to the `HPX::plugin` target. Similarly to `HPX::component` this will set `HPX_PLUGIN_NAME` to `hpx_<target_name>`. This definition should also be `PRIVATE`. Unlike regular shared libraries, plugins are loaded at runtime from certain directories and will not be found without additional configuration. Plugins should be installed into a directory containing only plugins. For example, the plugins created by *HPX* itself are installed into the `hpx` subdirectory in the library install directory (typically `lib` or `lib64`). When using the `HPX::plugin` target you need to install your plugins into an appropriate directory. You may also want to set the location of your plugin in the build directory with the `*_OUTPUT_DIRECTORY*` CMake target properties to be able to load the plugins in the build directory. Once you've set the install or output directory of your plugin you need to tell your executable where to find it at runtime. You can do this either by setting the environment variable `HPX_COMPONENT_PATHS` or the ini setting `hpx.component_paths` (see `--hpx:ini`) to the directory containing your plugin.

Using macros to create new targets

In addition to the targets described above, *HPX* provides convenience macros to hide optional boilerplate code that may be useful for your project. The link to the targets described above. We recommend that you use the targets directly whenever possible as they tend to compose better with other targets.

The macro for adding an *HPX* component is `add_hpx_component`. It can be used in your `CMakeLists.txt` file like this:

```
# build your application using HPX
add_hpx_component(hello_world
    SOURCES hello_world_component.cpp
    HEADERS hello_world_component.hpp
    COMPONENT_DEPENDENCIES iostreams)
```

Note: add_hpx_component adds a _component suffix to the target name. In the example above, a hello_world_component target will be created.

The available options to add_hpx_component are:

- SOURCES: The source files for that component
- HEADERS: The header files for that component
- DEPENDENCIES: Other libraries or targets this component depends on
- COMPONENT_DEPENDENCIES: The components this component depends on
- PLUGIN: Treats this component as a plugin-able library
- COMPILE_FLAGS: Additional compiler flags
- LINK_FLAGS: Additional linker flags
- FOLDER: Adds the headers and source files to this Source Group folder
- EXCLUDE_FROM_ALL: Do not build this component as part of the all target

After adding the component, the way you add the executable is as follows:

```
# build your application using HPX
add_hpx_executable(hello_world
    ESSENTIAL
    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[95] 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.

[95] 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:

```
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 `RPATH`s, 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[96].

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).

[96] 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.

```
// 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;
}
```

Copy the content of this program into a file called hello_world.cpp.

Now, in the directory where you put hello_world.cpp, create a Makefile. Add the following code:

```
CXX=(CXX)  # Add your favourite compiler here or let makefile choose default.

CXXFLAGS=-O3 -std=c++17

BOOST_ROOT=/path/to/boost
HWLOC_ROOT=/path/to/hwloc
TCMALLOC_ROOT=/path/to/tcmalloc
HPX_ROOT=/path/to/hpx

INCLUDE_DIRECTIVES=$(HPX_ROOT)/include $(BOOST_ROOT)/include $(HWLOC_ROOT)/include

LIBRARY_DIRECTIVES=-L$(HPX_ROOT)/lib $(HPX_ROOT)/lib/libhpx_init.a $(HPX_ROOT)/lib/
↪libhpx.so $(BOOST_ROOT)/lib/libboost_atomic-mt.so $(BOOST_ROOT)/lib/libboost_
↪filesystem-mt.so $(BOOST_ROOT)/lib/libboost_program_options-mt.so $(BOOST_ROOT)/lib/
↪libboost_regex-mt.so $(BOOST_ROOT)/lib/libboost_system-mt.so -lpthread $(TCMALLOC_
↪ROOT)/libtcmalloc_minimal.so $(HWLOC_ROOT)/libhwloc.so -ldl -lrt

LINK_FLAGS=$(HPX_ROOT)/lib/libhpx_wrap.a -Wl,-wrap=main  # should be left empty for␣
↪HPX_WITH_HPX_MAIN=OFF

hello_world: hello_world.o
   $(CXX) $(CXXFLAGS) -o hello_world hello_world.o $(LIBRARY_DIRECTIVES) $(LINK_FLAGS)

hello_world.o:
   $(CXX) $(CXXFLAGS) -c -o hello_world.o hello_world.cpp $(INCLUDE_DIRECTIVES)
```

Important: `LINK_FLAGS` should be left empty if HPX_WITH_HPX_MAIN is set to OFF. Boost in the above example is build with `--layout=tagged`. Actual Boost flags may vary on your build of Boost.

To build the program, type:

```
make
```

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

```
./hello_world
```

How to build *HPX* components with makefile

Let's try a more complex example involving an *HPX* component. An *HPX* component is a class that exposes *HPX* actions. *HPX* components are compiled into dynamically-loaded modules called component libraries. Here's the source code:

hello_world_component.cpp

```
#include "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);
```

hello_world_component.hpp

```
#pragma once

#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);
    };
}}
```

(continues on next page)

(continued from previous page)

```
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 "hello_world_component.hpp"
#include <hpx/hpx_main.hpp>

int main(int argc, char* argv[])
{
    {
        // 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;
}
```

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
```

(continues on next page)

(continued from previous page)

```
LIBRARY_DIRECTIVES=-L$(HPX_ROOT)/lib $(HPX_ROOT)/lib/libhpx_init.a $(HPX_ROOT)/lib/
↪libhpx.so $(BOOST_ROOT)/lib/libboost_atomic-mt.so $(BOOST_ROOT)/lib/libboost_
↪filesystem-mt.so $(BOOST_ROOT)/lib/libboost_program_options-mt.so $(BOOST_ROOT)/lib/
↪libboost_regex-mt.so $(BOOST_ROOT)/lib/libboost_system-mt.so -lpthread $(TCMALLOC_
↪ROOT)/libtcmalloc_minimal.so $(HWLOC_ROOT)/libhwloc.so -ldl -lrt

LINK_FLAGS=$(HPX_ROOT)/lib/libhpx_wrap.a -Wl,-wrap=main  # should be left empty for␣
↪HPX_WITH_HPX_MAIN=OFF

hello_world_client: libhpx_hello_world hello_world_client.o
  $(CXX) $(CXXFLAGS) -o hello_world_client $(LIBRARY_DIRECTIVES) libhpx_hello_world
↪$(LINK_FLAGS)

hello_world_client.o: hello_world_client.cpp
  $(CXX) $(CXXFLAGS) -o hello_world_client.o hello_world_client.cpp $(INCLUDE_
↪DIRECTIVES)

libhpx_hello_world: hello_world_component.o
  $(CXX) $(CXXFLAGS) -o libhpx_hello_world hello_world_component.o $(LIBRARY_
↪DIRECTIVES)

hello_world_component.o: hello_world_component.cpp
  $(CXX) $(CXXFLAGS) -c -o hello_world_component.o hello_world_component.cpp
↪$(INCLUDE_DIRECTIVES)
```

To build the program, type:

```
make
```

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

```
./hello_world
```

Note: Due to high variations in CMake flags and library dependencies, it is recommended to build *HPX* applications and components with pkg-config or CMakeLists.txt. Writing Makefile may result in broken builds if due care is not taken. pkg-config files and CMake systems are configured with CMake build of *HPX*. Hence, they are stable when used together and provide better support overall.

2.5.4 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:

```
#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 `argc`/`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:

```
#define main hpx_startup::user_main
```

which could result in unexpected behavior.

Important: To achieve this seamless integration, we use different implementations for different operating systems. In case of Linux or macOS, the code present in `hpx_wrap.cpp` is put into action. We hook into the system function in case of Linux and provide alternate entry point in case of macOS. For other operating systems we rely on a macro:

```
#define main hpx_startup::user_main
```

provided in the header file hpx/hpx_main.hpp. This implementation can result in unexpected behavior.

Caution: We make use of an *override* variable include_libhpx_wrap in the header file hpx/hpx_main.hpp to swiftly choose the function call stack at runtime. Therefore, the header file should *only* be included in the main executable. Including it in the components will result in multiple definition of the variable.

Supply your own main *HPX* entry point while blocking the main thread

With this method you need to provide an explicit main thread function named hpx_main at global scope. This function will be invoked as the main entry point of your *HPX* application on the console *locality* only (this function will be invoked as the first *HPX* thread of your application). All *HPX* API functions can be used from within this function.

The thread executing the function *hpx::init* will block waiting for the runtime system to exit. The value returned from hpx_main will be returned from *hpx::init* after the runtime system has stopped.

The function *hpx::finalize* has to be called on one of the *HPX* localities in order to signal that all work has been scheduled and the runtime system should be stopped after the scheduled work has been executed.

This method of invoking *HPX* has the advantage of 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.

```
#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:

```
int hpx_main();
int hpx_main(int argc, char* argv[]);
int hpx_main(hpx::program_options::variables_map& vm);
```

This is consistent with (and extends) the usually allowed prototypes for the function main() in C++ applications.

The header file to include for this method of using *HPX* is hpx/hpx_init.hpp.

There are many additional overloads of *hpx::init* available, such as 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 a 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.

```
#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:

```
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`.

```
#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.

HPX also supports suspending individual thread pools and threads. For details on how to do that see the documentation for *hpx::threads::thread_pool_base*.

Automatically suspending worker threads

The previous method guarantees that the worker threads are suspended when you ask for it and that they stay suspended. An alternative way to achieve the same effect is to tweak how quickly *HPX* suspends its worker threads when they run out of work. The following configuration values make sure that *HPX* idles very quickly:

```
hpx.max_idle_backoff_time = 1000
hpx.max_idle_loop_count = 0
```

They can be set on the command line using `--hpx:ini=hpx.max_idle_backoff_time=1000` and `--hpx:ini=hpx.max_idle_loop_count=0`. See *Launching and configuring HPX applications* for more details on how to set configuration parameters.

After setting idling parameters the previous example could now be written like this instead:

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

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

   // Initialize HPX, don't run hpx_main
    hpx::start(nullptr, argc, argv);

    // Schedule some functions on the HPX runtime
    // NOTE: run_as_hpx_thread blocks until completion.
    hpx::run_as_hpx_thread(&my_function, ...);
    hpx::run_as_hpx_thread(&my_other_function, ...);

    // hpx::finalize has to be called from the HPX runtime before hpx::stop
    hpx::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.

2.5.5 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[97]. 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.

[97] https://en.wikipedia.org/wiki/INI_file

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.

Built-in Default Configuration Settings

During startup any *HPX* application applies a predefined search pattern to locate one or more configuration files. All found files will be read and merged in the sequence they are found into one single internal data structure holding all configuration properties.

As a first step the internal configuration database is filled with a set of default configuration properties. Those settings are described on a section by section basis below.

Note: You can print the default configuration settings used for an executable by specifying the command line option *--hpx:dump-config*.

The `system` configuration section

```
[system]
pid = <process-id>
prefix = <current prefix path of core HPX library>
executable = <current prefix path of executable>
```

Property	Description
`system.pid`	This is initialized to store the current OS-process id of the application instance.
`system.prefix`	This is initialized to the base directory *HPX* has been loaded from.
`system.executable_prefix`	This is initialized to the base directory the current executable has been loaded from.

The `hpx` configuration section

```
[hpx]
location = ${HPX_LOCATION:$[system.prefix]}
component_path = $[hpx.location]/lib/hpx:$[system.executable_prefix]/lib/hpx:$[system.
↪executable_prefix]/../lib/hpx
master_ini_path = $[hpx.location]/share/hpx-<version>:$[system.executable_prefix]/
↪share/hpx-<version>:$[system.executable_prefix]/../share/hpx-<version>
ini_path = $[hpx.master_ini_path]/ini
os_threads = 1
localities = 1
program_name =
cmd_line =
lock_detection = ${HPX_LOCK_DETECTION:0}
throw_on_held_lock = ${HPX_THROW_ON_HELD_LOCK:1}
minimal_deadlock_detection = <debug>
spinlock_deadlock_detection = <debug>
spinlock_deadlock_detection_limit = ${HPX_SPINLOCK_DEADLOCK_DETECTION_LIMIT:1000000}
max_background_threads = ${HPX_MAX_BACKGROUND_THREADS:$[hpx.os_threads]}
max_idle_loop_count = ${HPX_MAX_IDLE_LOOP_COUNT:<hpx_idle_loop_count_max>}
max_busy_loop_count = ${HPX_MAX_BUSY_LOOP_COUNT:<hpx_busy_loop_count_max>}
max_idle_backoff_time = ${HPX_MAX_IDLE_BACKOFF_TIME:<hpx_idle_backoff_time_max>}
exception_verbosity = ${HPX_EXCEPTION_VERBOSITY:2}

[hpx.stacks]
small_size = ${HPX_SMALL_STACK_SIZE:<hpx_small_stack_size>}
medium_size = ${HPX_MEDIUM_STACK_SIZE:<hpx_medium_stack_size>}
large_size = ${HPX_LARGE_STACK_SIZE:<hpx_large_stack_size>}
huge_size = ${HPX_HUGE_STACK_SIZE:<hpx_huge_stack_size>}
use_guard_pages = ${HPX_THREAD_GUARD_PAGE:1}
```

Property	Description
`hpx.location`	This is initialized to the id of the *locality* this application instance is running on.
`hpx.component_path`	Duplicates are discarded. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows).
`hpx.master_ini_path`	This is initialized to the list of default paths of the main hpx.ini configuration files. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows).
`hpx.ini_path`	This is initialized to the default path where *HPX* will look for more ini configuration files. This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or using `';'` (Windows).
`hpx.os_threads`	This setting reflects the number of OS-threads used for running *HPX*-threads. Defaults to number of detected cores (not hyperthreads/PUs).
`hpx.localities`	This setting reflects the number of localities the application is running on. Defaults to `1`.
`hpx.program_name`	This setting reflects the program name of the application instance. Initialized from the command line `argv[0]`.
`hpx.cmd_line`	This setting reflects the actual command line used to launch this application instance.
`hpx.lock_detection`	This setting verifies that no locks are being held while a *HPX* thread is suspended. This setting is applicable only if `HPX_WITH_VERIFY_LOCKS` is set during configuration in CMake.
`hpx.throw_on_held_lock`	This setting causes an exception if during lock detection at least one lock is being held while a *HPX* thread is suspended. This setting is applicable only if `HPX_WITH_VERIFY_LOCKS` is set during configuration in CMake. This setting has no effect if `hpx.lock_detection=0`.
`hpx.minimal_deadlock_detection`	This setting enables support for minimal deadlock detection for *HPX*-threads. By default this is set to `1` (for Debug builds) or to `0` (for Release, RelWithDebInfo, RelMinSize builds), this setting is effective only if `HPX_WITH_THREAD_DEADLOCK_DETECTION` is set during configuration in CMake.
`hpx.spinlock_deadlock_detection`	This setting verifies that spinlocks don't spin longer than specified using the `hpx.spinlock_deadlock_detection_limit`. This setting is applicable only if `HPX_WITH_SPINLOCK_DEADLOCK_DETECTION` is set during configuration in CMake. By default this is set to `1` (for Debug builds) or to `0` (for Release, RelWithDebInfo, RelMinSize builds).
`hpx.spinlock_deadlock_detection_limit`	This setting specifies the upper limit of allowed number of spins that spinlocks are allowed to perform. This setting is applicable only if `HPX_WITH_SPINLOCK_DEADLOCK_DETECTION` is set during configuration in CMake. By default this is set to `1000000`.
`hpx.max_background_threads`	This setting defines the number of threads in the scheduler which are used to execute background work. By default this is the same as the number of cores used for the scheduler.
`hpx.max_idle_loop_count`	By default this is defined by the preprocessor constant `HPX_IDLE_LOOP_COUNT_MAX`. This is an internal setting which you should change only if you know exactly what you are doing.
`hpx.max_busy_loop_count`	This setting defines the maximum value of the busy-loop counter in the scheduler. By default this is defined by the preprocessor constant `HPX_BUSY_LOOP_COUNT_MAX`. This is an internal setting which you should change only if you know exactly what you are doing.
`hpx.max_idle_backoff_time`	This setting defines the maximum time (in milliseconds) for the scheduler to sleep after being idle for `hpx.max_idle_loop_count` iterations. This setting is applicable only if `HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF` is set during configuration in CMake. 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.
`hpx.exception_verbosity`	This setting defines the verbosity of exceptions. Valid values are integers. A setting of `2` or higher prints all available information. A setting of `1` leaves out the build configuration and environment variables. A setting of `0` or lower prints only the description of the thrown exception and the file name, function, and line number where the exception was thrown. The default value is `2` or the value of the environment variable `HPX_EXCEPTION_VERBOSITY`.
`hpx.stacks.small_size`	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 `0x8000`). This value is used for all *HPX* threads by default, except for the thread running `hpx_main` (which runs on a large stack).
`hpx.stacks.`	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`).

The `hpx.threadpools` configuration section

```
[hpx.threadpools]
io_pool_size = ${HPX_NUM_IO_POOL_SIZE:2}
parcel_pool_size = ${HPX_NUM_PARCEL_POOL_SIZE:2}
timer_pool_size = ${HPX_NUM_TIMER_POOL_SIZE:2}
```

Property	Description
`hpx.threadpools.io_pool_size`	The value of this property defines the number of OS-threads created for the internal I/O thread pool.
`hpx.threadpools.parcel_pool_size`	The value of this property defines the number of OS-threads created for the internal parcel thread pool.
`hpx.threadpools.timer_pool_size`	The value of this property defines the number of OS-threads created for the internal timer thread pool.

The `hpx.thread_queue` configuration section

Important: These setting 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:10}
min_add_new_count = ${HPX_THREAD_QUEUE_MIN_ADD_NEW_COUNT:10}
max_add_new_count = ${HPX_THREAD_QUEUE_MAX_ADD_NEW_COUNT:10}
max_delete_count = ${HPX_THREAD_QUEUE_MAX_DELETE_COUNT:1000}
```

Property	Description
`hpx.thread_queue.min_tasks_to_steal_pending`	The value of this property defines the number of pending *HPX* threads which have to be available before neighboring cores are allowed to steal work. The default is to allow stealing always.
`hpx.thread_queue.min_tasks_to_steal_staged`	The value of this property defines the number of staged *HPX* tasks have which to be available before neighboring cores are allowed to steal work. The default is to allow stealing only if there are more tan 10 tasks available.
`hpx.thread_queue.min_add_new_count`	The value of this property defines the minimal number tasks to be converted into *HPX* threads whenever the thread queues for a core have run empty.
`hpx.thread_queue.max_add_new_count`	The value of this property defines the maximal number tasks to be converted into *HPX* threads whenever the thread queues for a core have run empty.
`hpx.thread_queue.max_delete_count`	The value of this property defines the number of terminated *HPX* threads to discard during each invocation of the corresponding function.

The `hpx.components` configuration section

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

Property	Description
`hpx.components.load_external`	This entry defines whether external components will be loaded on this *locality*. This entry 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*.

Additionally, the section `hpx.components` will be populated with the information gathered from all found components. The information loaded for each of the components will contain at least the following properties:

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

Property	Description
`hpx.components.<component_instance_name>.name`	This is the name of a component, usually the same as the second argument to the macro used while registering the component with *`HPX_REGISTER_COMPONENT`*. Set by the component factory.
`hpx.components.<component_instance_name>.path`	This is either the full path file name of the component module or the directory the component module is located in. In this case, the component module name will be derived from the property `hpx.components.<component_instance_name>.name`. Set by the component factory.
`hpx.components.<component_instance_name>.enabled`	This setting explicitly enables or disables the component. This is an optional property, *HPX* assumed that the component is enabled if it is not defined.

The value for `<component_instance_name>` is usually the same as for the corresponding `name` property. However generally it can be defined to any arbitrary instance name. It is used to distinguish between different ini sections, one for each component.

The `hpx.parcel` configuration section

```
[hpx.parcel]
address = ${HPX_PARCEL_SERVER_ADDRESS:<hpx_initial_ip_address>}
port = ${HPX_PARCEL_SERVER_PORT:<hpx_initial_ip_port>}
bootstrap = ${HPX_PARCEL_BOOTSTRAP:<hpx_parcel_bootstrap>}
max_connections = ${HPX_PARCEL_MAX_CONNECTIONS:<hpx_parcel_max_connections>}
max_connections_per_locality = ${HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY:<hpx_parcel_
↪max_connections_per_locality>}
max_message_size = ${HPX_PARCEL_MAX_MESSAGE_SIZE:<hpx_parcel_max_message_size>}
max_outbound_message_size = ${HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE:<hpx_parcel_max_
↪outbound_message_size>}
array_optimization = ${HPX_PARCEL_ARRAY_OPTIMIZATION:1}
zero_copy_optimization = ${HPX_PARCEL_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.array_
↪optimization]}
```

(continues on next page)

(continued from previous page)

```
async_serialization = ${HPX_PARCEL_ASYNC_SERIALIZATION:1}
message_handlers = ${HPX_PARCEL_MESSAGE_HANDLERS:0}
```

Property	Description
hpx.parcel.address	This property defines the default IP address to be used for the *parcel* layer to listen to. This IP address will be used as long as no other values are specified (for instance using the *--hpx:hpx* command line option). The expected format is any valid IP address or domain name format which can be resolved into an IP address. The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_ADDRESS` ("127.0.0.1").
hpx.parcel.port	This property defines the default IP port to be used for the *parcel* layer to listen to. This IP port will be used as long as no other values are specified (for instance using the *--hpx:hpx* command line option). The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_PORT` (7910).
hpx.parcel.bootstrap	This property defines which parcelport type should be used during application bootstrap. The default depends on the compile time preprocessor constant `HPX_PARCEL_BOOTSTRAP` ("tcp").
hpx.parcel.max_connections	This property defines how many network connections between different localities are overall kept alive by each of *locality*. The default depends on the compile time preprocessor constant `HPX_PARCEL_MAX_CONNECTIONS` (512).
hpx.parcel.max_connections_per_locality	This property defines the maximum number of network connections that one *locality* will open to another *locality*. The default depends on the compile time preprocessor constant `HPX_PARCEL_MAX_CONNECTIONS_PER_LOCALITY` (4).
hpx.parcel.max_message_size	This property defines the maximum allowed message size which will be transferrable through the *parcel* layer. The default depends on the compile time preprocessor constant `HPX_PARCEL_MAX_MESSAGE_SIZE` (1000000000 bytes).
hpx.parcel.max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size which will be transferrable through the parcel layer. The default depends on the compile time preprocessor constant `HPX_PARCEL_MAX_OUTBOUND_MESSAGE_SIZE` (1000000 bytes).
hpx.parcel.array_optimization	This property defines whether this *locality* is allowed to utilize array optimizations during serialization of *parcel* data. The default is 1.
hpx.parcel.zero_copy_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations during serialization of *parcel* data. The default is the same value as set for `hpx.parcel.array_optimization`.
hpx.parcel.async_serialization	This property defines whether this *locality* is allowed to spawn a new thread for serialization (this is both for encoding and decoding parcels). The default is 1.
hpx.parcel.message_handlers	This property defines whether message handlers are loaded. The default is 0.

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 = ${HPX_PARCEL_TCP_ASYNC_SERIALIZATION:$[hpx.parcel.async_
↪serialization]}
parcel_pool_size = ${HPX_PARCEL_TCP_PARCEL_POOL_SIZE:$[hpx.threadpools.parcel_pool_
↪size]}
```

(continues on next page)

(continued from previous page)

```
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]}
```

Property	Description
`hpx.parcel.tcp.enable`	Enable the use of the default TCP parcelport. Note that the initial bootstrap of the overall *HPX* application will be performed using the default TCP connections. This parcelport is enabled by default. This will be disabled only if MPI is enabled (see below).
`hpx.parcel.tcp.array_optimization`	This property defines whether this *locality* is allowed to utilize array optimizations in the TCP/IP parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.array_optimization`.
`hpx.parcel.tcp.zero_copy_optimization`	This property defines whether this *locality* is allowed to utilize zero copy optimizations in the TCP/IP parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`.
`hpx.parcel.tcp.async_serialization`	This property defines whether this *locality* is allowed to spawn a new thread for serialization in the TCP/IP parcelport (this is both for encoding and decoding parcels). The default is the same value as set for `hpx.parcel.async_serialization`.
`hpx.parcel.tcp.parcel_pool_size`	The value of this property defines the number of OS-threads created for the internal parcel thread pool of the TCP *parcel* port. The default is taken from `hpx.threadpools.parcel_pool_size`.
`hpx.parcel.tcp.max_connections`	This property defines how many network connections between different localities are overall kept alive by each of *locality*. The default is taken from `hpx.parcel.max_connections`.
`hpx.parcel.tcp.max_connections_per_locality`	This property defines the maximum number of network connections that one *locality* will open to another *locality*. The default is taken from `hpx.parcel.max_connections_per_locality`.
`hpx.parcel.tcp.max_message_size`	This property defines the maximum allowed message size which will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_message_size`.
`hpx.parcel.tcp.max_outbound_message_size`	This property defines the maximum allowed outbound coalesced message size which will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_outbound_connections`.

The following settings relate to the MPI parcelport. These settings take effect only if the compile time constant `HPX_HAVE_PARCELPORT_MPI` is set (the equivalent cmake variable is `HPX_WITH_PARCELPORT_MPI` and has to be set to `ON`.

```
[hpx.parcel.mpi]
enable = ${HPX_HAVE_PARCELPORT_MPI:$[hpx.parcel.enabled]}
env = ${HPX_HAVE_PARCELPORT_MPI_ENV:MV2_COMM_WORLD_RANK,PMI_RANK,OMPI_COMM_WORLD_SIZE,
↪ALPS_APP_PE}
multithreaded = ${HPX_HAVE_PARCELPORT_MPI_MULTITHREADED:0}
rank = <MPI_rank>
processor_name = <MPI_processor_name>
array_optimization = ${HPX_HAVE_PARCEL_MPI_ARRAY_OPTIMIZATION:$[hpx.parcel.array_
↪optimization]}
zero_copy_optimization = ${HPX_HAVE_PARCEL_MPI_ZERO_COPY_OPTIMIZATION:$[hpx.parcel.
↪zero_copy_optimization]}
use_io_pool = ${HPX_HAVE_PARCEL_MPI_USE_IO_POOL:$1}
```

(continues on next page)

(continued from previous page)

```
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]}
```

Property	Description
hpx.parcel.mpi.enable	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.
hpx.parcel.mpi.env	This property influences which environment variables (comma separated) will be analyzed to find out whether the application was invoked by MPI.
hpx.parcel.mpi.multithreaded	This property is used to determine what threading mode to use when initializing MPI. If this setting is `0` *HPX* will initialize MPI with `MPI_THREAD_SINGLE` if the value is not equal to `0` *HPX* will initialize MPI with `MPI_THREAD_MULTI`.
hpx.parcel.mpi.rank	This property will be initialized to the MPI rank of the *locality*.
hpx.parcel.mpi.processor_name	This property will be initialized to the MPI processor name of the *locality*.
hpx.parcel.mpi.array_optimization	This property defines whether this *locality* is allowed to utilize array optimizations in the MPI parcelport during serialization of *parcel* data. The default is the same value as set for `hpx.parcel.array_optimization`.
hpx.parcel.mpi.zero_copy_optimization	This property defines whether this *locality* is allowed to utilize zero copy optimizations in the MPI parcelport during serialization of parcel data. The default is the same value as set for `hpx.parcel.zero_copy_optimization`.
hpx.parcel.mpi.use_io_pool	This property can be set to run the progress thread inside of HPX threads instead of a separate thread pool. The default is `1`.
hpx.parcel.mpi.async_serialization	This property defines whether this *locality* is allowed to spawn a new thread for serialization in the MPI parcelport (this is both for encoding and decoding parcels). The default is the same value as set for `hpx.parcel.async_serialization`.
hpx.parcel.mpi.parcel_pool_size	The value of this property defines the number of OS-threads created for the internal parcel thread pool of the MPI *parcel* port. The default is taken from `hpx.threadpools.parcel_pool_size`.
hpx.parcel.mpi.max_connections	This property defines how many network connections between different localities are overall kept alive by each of *locality*. The default is taken from `hpx.parcel.max_connections`.
hpx.parcel.mpi.max_connections_per_locality	This property defines the maximum number of network connections that one *locality* will open to another *locality*. The default is taken from `hpx.parcel.max_connections_per_locality`.
hpx.parcel.mpi.max_message_size	This property defines the maximum allowed message size which will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_message_size`.
hpx.parcel.mpi.max_outbound_message_size	This property defines the maximum allowed outbound coalesced message size which will be transferrable through the *parcel* layer. The default is taken from `hpx.parcel.max_outbound_connections`.

The hpx.agas configuration section

```
[hpx.agas]
address = ${HPX_AGAS_SERVER_ADDRESS:<hpx_initial_ip_address>}
port = ${HPX_AGAS_SERVER_PORT:<hpx_initial_ip_port>}
service_mode = hosted
dedicated_server = 0
max_pending_refcnt_requests = ${HPX_AGAS_MAX_PENDING_REFCNT_REQUESTS:<hpx_initial_
↪agas_max_pending_refcnt_requests>}
use_caching = ${HPX_AGAS_USE_CACHING:1}
use_range_caching = ${HPX_AGAS_USE_RANGE_CACHING:1}
local_cache_size = ${HPX_AGAS_LOCAL_CACHE_SIZE:<hpx_agas_local_cache_size>}
```

Property	Description
`hpx.agas.address`	This property defines the default IP address to be used for the *AGAS* root server. This IP address will be used as long as no other values are specified (for instance using the *--hpx:agas* command line option). The expected format is any valid IP address or domain name format which can be resolved into an IP address. The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_ADDRESS` ("127.0.0.1").
`hpx.agas.port`	This property defines the default IP port to be used for the *AGAS* root server. This IP port will be used as long as no other values are specified (for instance using the *--hpx:agas* command line option). The default depends on the compile time preprocessor constant `HPX_INITIAL_IP_PORT` (7009).
`hpx.agas.service_mode`	This property specifies what type of *AGAS* service is running on this *locality*. Currently, two modes exist. The *locality* that acts as the *AGAS* server runs in `bootstrap` mode. All other localities are in `hosted` mode.
`hpx.agas.dedicated_server`	This property specifies whether the *AGAS* server is exclusively running *AGAS* services and not hosting any application components. It is a boolean value. Set to `1` if *-hpx:run-agas-server-only* is present.
`hpx.agas.max_pending_refcnt_requests`	This property defines the number of reference counting requests (increments or decrements) to buffer. The default depends on the compile time preprocessor constant `HPX_INITIAL_AGAS_MAX_PENDING_REFCNT_REQUESTS` (4096).
`hpx.agas.use_caching`	This property specifies whether a software address translation cache is used. It is a boolean value. Defaults to `1`.
`hpx.agas.use_range_caching`	This property specifies whether range-based caching is used by the software address translation cache. This property is ignored if *hpx.agas.use_caching* is false. It is a boolean value. Defaults to `1`.
`hpx.agas.local_cache_size`	This property defines the size of the software address translation cache for *AGAS* services. This property is ignored if `hpx.agas.use_caching` is false. Note that if `hpx.agas.use_range_caching` is true, this size will refer to the maximum number of ranges stored in the cache, not the number of entries spanned by the cache. The default depends on the compile time preprocessor constant `HPX_AGAS_LOCAL_CACHE_SIZE` (4096).

The `hpx.commandline` configuration section

The following table lists the definition of all pre-defined command line option shortcuts. For more information about commandline options see the section *HPX Command Line Options*.

```
[hpx.commandline]
aliasing = ${HPX_COMMANDLINE_ALIASING:1}
allow_unknown = ${HPX_COMMANDLINE_ALLOW_UNKNOWN:0}

[hpx.commandline.aliases]
-a = --hpx:agas
-c = --hpx:console
-h = --hpx:help
-I = --hpx:ini
-l = --hpx:localities
-p = --hpx:app-config
-q = --hpx:queuing
-r = --hpx:run-agas-server
-t = --hpx:threads
-v = --hpx:version
-w = --hpx:worker
-x = --hpx:hpx
-0 = --hpx:node=0
-1 = --hpx:node=1
-2 = --hpx:node=2
-3 = --hpx:node=3
-4 = --hpx:node=4
-5 = --hpx:node=5
-6 = --hpx:node=6
-7 = --hpx:node=7
-8 = --hpx:node=8
-9 = --hpx:node=9
```

Property	Description
hpx.commandline.aliasing	Enable command line aliases as defined in the section hpx.commandline.aliases (see below). Defaults to 1.
hpx.commandline.allow_unknown	Allow for unknown command line options to be passed through to hpx_main() Defaults to 0.
hpx.commandline.aliases.-a	On the commandline, -a expands to: *--hpx:agas*.
hpx.commandline.aliases.-c	On the commandline, -c expands to: *--hpx:console*.
hpx.commandline.aliases.-h	On the commandline, -h expands to: *--hpx:help*.
hpx.commandline.aliases.--help	On the commandline, --help expands to: *--hpx:help*.
hpx.commandline.aliases.-I	On the commandline, -I expands to: *--hpx:ini*.
hpx.commandline.aliases.-l	On the commandline, -l expands to: *--hpx:localities*.
hpx.commandline.aliases.-p	On the commandline, -p expands to: *--hpx:app-config*.
hpx.commandline.aliases.-q	On the commandline, -q expands to: *--hpx:queuing*.
hpx.commandline.aliases.-r	On the commandline, -r expands to: *--hpx:run-agas-server*.
hpx.commandline.aliases.-t	On the commandline, -t expands to: *--hpx:threads*.
hpx.commandline.aliases.-v	On the commandline, -v expands to: *--hpx:version*.
hpx.commandline.aliases.--version	On the commandline, --version expands to: *--hpx:version*.
hpx.commandline.aliases.-w	On the commandline, -w expands to: *--hpx:worker*.
hpx.commandline.aliases.-x	On the commandline, -x expands to: *--hpx:hpx*.
hpx.commandline.aliases.-0	On the commandline, -0 expands to: *--hpx:node*=0.
hpx.commandline.aliases.-1	On the commandline, -1 expands to: *--hpx:node*=1.
hpx.commandline.aliases.-2	On the commandline, -2 expands to: *--hpx:node*=2.
hpx.commandline.aliases.-3	On the commandline, -3 expands to: *--hpx:node*=3.
hpx.commandline.aliases.-4	On the commandline, -4 expands to: *--hpx:node*=4.
hpx.commandline.aliases.-5	On the commandline, -5 expands to: *--hpx:node*=5.
hpx.commandline.aliases.-6	On the commandline, -6 expands to: *--hpx:node*=6.
hpx.commandline.aliases.-7	On the commandline, -7 expands to: *--hpx:node*=7.
hpx.commandline.aliases.-8	On the commandline, -8 expands to: *--hpx:node*=8.
hpx.commandline.aliases.-9	On the commandline, -9 expands to: *--hpx:node*=9.

Loading INI files

During startup and after the internal database has been initialized as described in the section *Built-in Default Configuration Settings*, *HPX* will try to locate and load additional ini files to be used as a source for configuration properties. This allows for a wide spectrum of additional customization possibilities by the user and system administrators. The sequence of locations where *HPX* will try loading the ini files is well defined and documented in this section. All ini files found are merged into the internal configuration database. The merge operation itself conforms to the rules as described in the section *The HPX INI File Format*.

1. Load all component shared libraries found in the directories specified by the property `hpx.component_path` and retrieve their default configuration information (see section *Loading components* for more details). This property can refer to a list of directories separated by `':'` (Linux, Android, and MacOS) or 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:

```
[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`:

```
// 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:

```
// 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:

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

```

(continues on next page)

(continued from previous 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<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 simples 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`.

```
# file: $APP_ROOT/app_server.ini
[hpx.components.app_server]
name = app_server
path = $APP_LOCATION/
```

Here `$APP_LOCATION` is the directory where the (binary) component shared library is located. *HPX* will attempt to load the shared library from there. The section name `hpx.components.app_server` reflects the instance name of the component (`app_server` is an arbitrary, but unique name). The property value for `hpx.components.app_server.name` should be the same as used for the second argument to the macro *HPX_REGISTER_COMPONENT* above.

Additionally a file `.hpx.ini` which could be located in the current working directory (see step 3 as described in the section *Loading INI files*) can be used to add to the ini search path for components:

```
# file: $PWD/.hpx.ini
[hpx]
ini_path = $[hpx.ini_path]:$APP_ROOT/
```

This assumes that the above ini file specific to the component is located in the directory `$APP_ROOT`.

Note: It is possible to reference the defined property from inside its value. *HPX* will gracefully use the previous value of `hpx.ini_path` for the reference on the right hand side and assign the overall (now expanded) value to the property.

Logging

HPX uses a sophisticated logging framework allowing 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 2.7: Logging categories

Category	Category shortcut	Information to be generated	Environment variable
General	None	Logging information generated by different subsystems of *HPX*, such as thread-manager, parcel layer, LCOs, etc.	`HPX_LOGLEVEL`
AGAS	`AGAS`	Logging output generated by the *AGAS* subsystem	`HPX_AGAS_LOGLEVEL`
Application	`APP`	Logging generated by applications.	`HPX_APP_LOGLEVEL`

By default, all logging output is redirected to the console instance of an application, where it is collected and written to a file, one file for each logging category.

Each logging category can be customized at two levels, the parameters for each are stored in the ini configuration sections `hpx.logging.CATEGORY` and `hpx.logging.console.CATEGORY` (where `CATEGORY` is the category shortcut as listed in the table above). The former influences logging at the source *locality* and the latter modifies the logging behaviour for each of the categories at the console instance of an application.

Levels

All *HPX* logging output has seven different logging levels. These levels can be set explicitly or through 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 2.8: Logging levels

Logging level	Integral value
`<debug>`	5
`<info>`	4
`<warning>`	3
`<error>`	2
`<fatal>`	1
No logging	0

Tip: The easiest way to enable logging output is to set the environment variable corresponding to the logging category to an integral value as described in the table above. For instance, setting `HPX_LOGLEVEL=5` will enable full logging output for the general category. Please note that the syntax and means of setting environment variables varies between operating systems.

Configuration

Logs will be saved to destinations as configured by the user. By default, logging output is saved on the console instance of an application to `hpx.<CATEGORY>.<PID>.lo` (where `CATEGORY` and `PID>` are placeholders for the category shortcut and the OS process id). The output for the general logging category is saved to `hpx.<PID>.log`. The default settings for the general logging category are shown here (the syntax is described in the section *The HPX INI File Format*):

```
[hpx.logging]
level = ${HPX_LOGLEVEL:0}
destination = ${HPX_LOGDESTINATION:console}
format = ${HPX_LOGFORMAT:(T%locality%/%hpxthread%.%hpxphase%/%hpxcomponent%) P
↪%parentloc%/%hpxparent%.%hpxparentphase% %time%($hh:$mm.$ss.$mili) [%idx%]|\\n}
```

The logging level is taken from the environment variable `HPX_LOGLEVEL` and defaults to zero, e.g. no logging. The default logging destination is read from the environment variable `HPX_LOGDESTINATION` On any of the localities it defaults to `console` which redirects all generated logging output to the console instance of an application. The following table lists the possible destinations for any logging output. It is possible to specify more than one destination separated by whitespace.

Table 2.9: Logging destinations

Logging destination	Description
file(`<filename>`)	Direct all output to a file with the given <filename>.
cout	Direct all output to the local standard output of the application instance on this *locality*.
cerr	Direct all output to the local standard error output of the application instance on this *locality*.
console	Direct all output to the console instance of the application. The console instance has its logging destinations configured separately.
android_log	Direct all output to the (Android) system log (available on Android systems only).

The logging format is read from the environment variable `HPX_LOGFORMAT` and it defaults to a complex format description. This format consists of several placeholder fields (for instance `%locality%` which will be replaced by concrete values when the logging output is generated. All other information is transferred verbatim to the output. The table below describes the available field placeholders. The separator character `|` separates the logging message prefix formatted as shown and the actual log message which will replace the separator.

Table 2.10: Available field placeholders

Name	Description
locality	The id of the *locality* on which the logging message was generated.
hpxthread	The id of the *HPX*-thread generating this logging output.
hpxphase	The phase[99] of the *HPX*-thread generating this logging output.
hpxcomponent	The local virtual address of the component which the current *HPX*-thread is accessing.
parentloc	The id of the *locality* where the *HPX* thread was running which initiated the current *HPX*-thread. The current *HPX*-thread is generating this logging output.
hpxparent	The id of the *HPX*-thread which initiated the current *HPX*-thread. The current *HPX*-thread is generating this logging output.
hpxparentphase	The phase of the *HPX*-thread when it initiated the current *HPX*-thread. The current *HPX*-thread is generating this logging output.
time	The time stamp for this logging outputline as generated by the source *locality*.
idx	The sequence number of the logging output line as generated on the source *locality*.
osthread	The sequence number of the OS-thread which executes the current *HPX*-thread.

Note: Not all of the field placeholder may be expanded for all generated logging output. If no value is available for a particular field it is replaced with a sequence of `'-'` characters.]

Here is an example line from a logging output generated by one of the *HPX* examples (please note that this is generated on a single line, without line break):

```
(T00000000/0000000002d46f90.01/00000000009ebc10) P--------/0000000002d46f80.02 17:49.
↪37.320 [000000000000004d]
    <info>  [RT] successfully created component {0000000100ff0001, 0000000000030002}␣
↪of type: component_barrier[7(3)]
```

The default settings for the general logging category on the console is shown here:

```
[hpx.logging.console]
level = ${HPX_LOGLEVEL:$[hpx.logging.level]}
destination = ${HPX_CONSOLE_LOGDESTINATION:file(hpx.$[system.pid].log)}
format = ${HPX_CONSOLE_LOGFORMAT:|}
```

These settings define how the logging is customized once the logging output is received by the console instance of an application. The logging level is read from the environment variable `HPX_LOGLEVEL` (as set for the console instance of the application). The level defaults to the same values as the corresponding settings in the general logging configuration shown before. The destination on the console instance is set to be a file 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.

[99] The phase of a *HPX*-thread counts how often this thread has been activated.

HPX Command Line Options

The predefined command line options for any application using *hpx::init* are described in the following subsections.

HPX options (allowed on command line only)

--hpx:help
print out program usage (default: this message), possible values: `full` (additionally prints options from components)

--hpx:version
print out *HPX* version and copyright information

--hpx:info
print out *HPX* configuration information

--hpx:options-file `arg`
specify a file containing command line options (alternatively: @filepath)

HPX options (additionally allowed in an options file)

--hpx:worker
run this instance in worker mode

--hpx:console
run this instance in console mode

--hpx:connect
run this instance in worker mode, but connecting late

--hpx:run-agas-server
run *AGAS* server as part of this runtime instance

--hpx:run-hpx-main
run the hpx_main function, regardless of *locality* mode

--hpx:hpx `arg`
the IP address the *HPX* parcelport is listening on, expected format: `address:port` (default: `127.0.0.1:7910`)

--hpx:agas `arg`
the IP address the *AGAS* root server is running on, expected format: `address:port` (default: `127.0.0.1:7910`)

--hpx:run-agas-server-only
run only the *AGAS* server

--hpx:nodefile `arg`
the file name of a node file to use (list of nodes, one node name per line and core)

--hpx:nodes `arg`
the (space separated) list of the nodes to use (usually this is extracted from a node file)

--hpx:endnodes
this can be used to end the list of nodes specified using the option *--hpx:nodes*

--hpx:ifsuffix `arg`
suffix to append to host names in order to resolve them to the proper network interconnect

--hpx:ifprefix `arg`
prefix to prepend to host names in order to resolve them to the proper network interconnect

--hpx:iftransform `arg`
sed-style search and replace (`s/search/replace/`) used to transform host names to the proper network interconnect

--hpx: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 2.11: Predefined command line option shortcuts

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

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:

```
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:` defines that for each of the cores (`core:0-3`) only their first processing unit `pu:0` should be used.

Note: The command line option `--hpx:print-bind` can be used to print the bitmasks generated from the affinity mappings as specified with `--hpx:bind`. For instance, on a system with hyperthreading enabled (i.e. 2 processing units per core), the command line:

```
hello_world_distributed -t4 --hpx:bind=thread:0-3=core:0-3.pu:0 --hpx:print-bind
```

will cause this output to be printed:

```
0: PU L#0(P#0), Core L#0, Socket L#0, Node L#0(P#0)
1: PU L#2(P#2), Core L#1, Socket L#0, Node L#0(P#0)
2: PU L#4(P#4), Core L#2, Socket L#0, Node L#0(P#0)
3: PU L#6(P#6), Core L#3, Socket L#0, Node L#0(P#0)
```

where each bit in the bitmasks corresponds to a processing unit the listed worker thread will be bound to run on.

The difference between the four possible predefined distribution schemes (`compact`, `scatter`, `balanced` and `numa-balanced`) is best explained with an example. Imagine that we have a system with 4 cores and 4 hardware threads per core on 2 sockets. If we place 8 threads the assignments produced by the `compact`, `scatter`, `balanced` and `numa-balanced` types are shown in the figure below. Notice that `compact` does not fully utilize all the cores in the system. For this reason it is recommended that applications are run using the `scatter` or `balanced`/`numa-balanced` options in most cases.

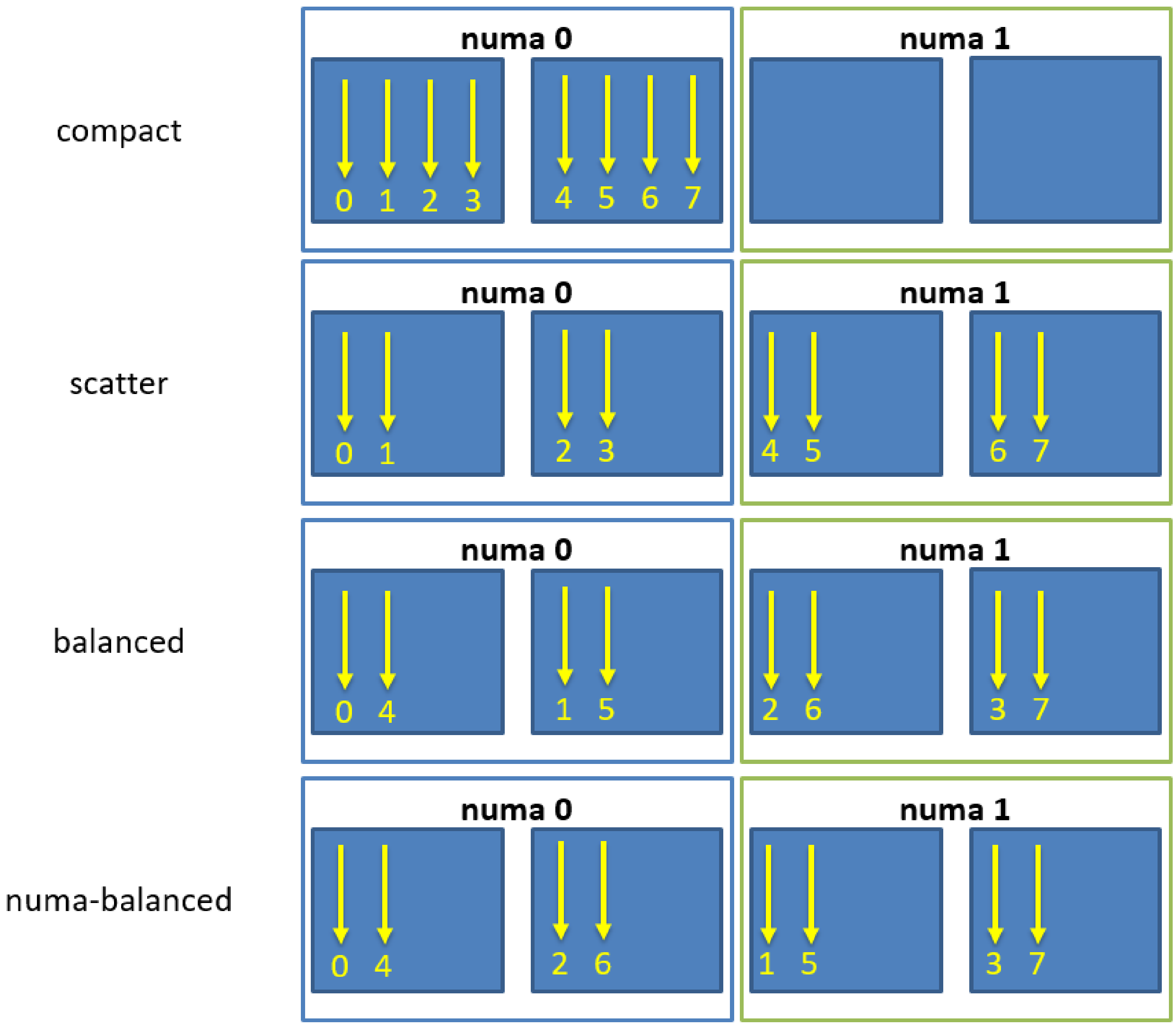

Fig. 2.7: Schematic of thread affinity type distributions.

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[98] 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.

2.5.6 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 Object*s (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:

```
    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());
    hpx::cout << f.get() << hpx::endl;
```

Channels can be handed to another thread (or in case of channel components, to other localities), thus establishing a communication channel between two independent places in the program:

```
void do_something(hpx::lcos::local::receive_channel<int> c,
    hpx::lcos::local::send_channel<> done)
{
    // prints 43
    hpx::cout << c.get(hpx::launch::sync) << hpx::endl;
    // signal back
    done.set();
}
```

(continues on next page)

98 https://en.wikipedia.org/wiki/Message_Passing_Interface

(continued from previous page)

```
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:

```
// channel components need to be registered for each used type (not needed
// for hpx::lcos::local::channel)
HPX_REGISTER_CHANNEL(double);

void channel_sender(hpx::lcos::channel<double> c)
{
    for (double d : c)
        hpx::cout << d << std::endl;
}
HPX_PLAIN_ACTION(channel_sender);

void channel()
{
    // create the channel on this locality
    hpx::lcos::channel<double> c(hpx::find_here());

    // pass the channel to a (possibly remote invoked) action
    hpx::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):

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

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

```
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:

```
#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 2.13: Facilities extending `std::future`

Facility	Description
`hpx::future::then`	In asynchronous programming, it is very common for one asynchronous operation, on completion, to invoke a second operation and pass data to it. The current C++ standard does not allow one to register a continuation to a future. With `then`, instead of waiting for the result, a continuation is "attached" to the asynchronous operation, which is invoked when the result is ready. Continuations registered using then function will help to avoid blocking waits or wasting threads on polling, greatly improving the responsiveness and scalability of an application.
unwrapping constructor for `hpx::future`	In some scenarios, you might want to create a future that returns another future, resulting in nested futures. Although it is possible to write code to unwrap the outer future and retrieve the nested future and its result, such code is not easy to write because users must handle exceptions and it may cause a blocking call. Unwrapping can allow users to mitigate this problem by doing an asynchronous call to unwrap the outermost future.
`hpx::future::is_ready`	There are often situations where a `get()` call on a future may not be a blocking call, or is only a blocking call under certain circumstances. This function gives the ability to test for early completion and allows us to avoid associating a continuation, which needs to be scheduled with some non-trivial overhead and near-certain loss of cache efficiency.
`hpx::make_ready_future`	Some functions may know the value at the point of construction. In these cases the value is immediately available, but needs to be returned as a future. By using `hpx::make_ready_future` a future can be created that holds a pre-computed result in its shared state. In the current standard it is non-trivial to create a future directly from a value. First a promise must be created, then the promise is set, and lastly the future is retrieved from the promise. This can now be done with one operation.

The standard also omits the ability to compose multiple futures. This is a common pattern that is ubiquitous in other asynchronous frameworks and is absolutely necessary in order to make C++ a powerful asynchronous programming language. Not including these functions is synonymous to Boolean algebra without AND/OR.

In addition to the extensions proposed by N4313[100], *HPX* adds functions allowing users to compose several futures in a more flexible way.

[100] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html

Table 2.14: Facilities for composing `hpx::futures`

Facility	Description	Comment
hpx::when_any, *hpx::when_any_n*	Asynchronously wait for at least one of multiple future or shared_future objects to finish.	N4313[101], `..._n` versions are *HPX* only
hpx::wait_any, *hpx::wait_any_n*	Synchronously wait for at least one of multiple future or shared_future objects to finish.	*HPX* only
hpx::when_all, *hpx::when_all_n*	Asynchronously wait for all future and shared_future objects to finish.	N4313[102], `..._n` versions are *HPX* only
hpx::wait_all, *hpx::wait_all_n*	Synchronously wait for all future and shared_future objects to finish.	*HPX* only
hpx::when_some, *hpx::when_some_n*	Asynchronously wait for multiple future and shared_future objects to finish.	*HPX* only
hpx::wait_some, *hpx::wait_some_n*	Synchronously wait for multiple future and shared_future objects to finish.	*HPX* only
hpx::when_each	Asynchronously wait for multiple future and shared_future objects to finish and call a function for each of the future objects as soon as it becomes ready.	*HPX* only
hpx::wait_each, *hpx::wait_each_n*	Synchronously wait for multiple future and shared_future objects to finish and call a function for each of the future objects as soon as it becomes ready.	*HPX* only

High level parallel facilities

In preparation for the upcoming C++ Standards, there are currently several proposals targeting different facilities supporting parallel programming. *HPX* implements (and extends) some of those proposals. This is well aligned with our strategy to align the APIs exposed from *HPX* with current and future C++ Standards.

At this point, *HPX* implements several of the C++ Standardization working papers, most notably N4409[103] (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411[104] (Task Blocks), and N4406[105] (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

[101] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
[102] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
[103] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf
[104] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
[105] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.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::parallel::execution::sequenced_policy`* or *`hpx::parallel::execution::sequenced_task_policy`* execute in sequential order. For *`hpx::parallel::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::parallel::execution::parallel_policy`* or *`hpx::parallel::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::parallel::execution::parallel_unsequenced_policy`* is, in *HPX*, equivalent to the use of the execution policy *`hpx::parallel::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::parallel::execution::sequenced_task_policy`* or *`hpx::parallel::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::parallel::execution::sequenced_policy`* or *`hpx::parallel::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::parallel::execution::parallel_unsequenced_policy`*, `hpx::terminate` shall be called.
- If the execution policy object is of type *`hpx::parallel::execution::sequenced_policy`*, *`hpx::parallel::execution::sequenced_task_policy`*, *`hpx::parallel::execution::parallel_pol`* or *`hpx::parallel::execution::parallel_task_policy`*, the execution of the algorithm terminates with an *`hpx::exception_list`* exception. All uncaught exceptions thrown during the application of user-provided function objects shall be contained in the *`hpx::exception_list`*.

For example, the number of invocations of the user-provided function object in for_each is unspecified. When `hpx::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 2.15: Non-modifying parallel algorithms (in header: `<hpx/algorithm.hpp>`)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::adjacent_find	Computes the differences between adjacent elements in a range.	`<hpx/algorithm.hpp>`	adjacent_find[106]
hpx::all_of	Checks if a predicate is `true` for all of the elements in a range.	`<hpx/algorithm.hpp>`	all_any_none_of[107]
hpx::any_of	Checks if a predicate is `true` for any of the elements in a range.	`<hpx/algorithm.hpp>`	all_any_none_of[108]
hpx::count	Returns the number of elements equal to a given value.	`<hpx/algorithm.hpp>`	count[109]
hpx::count_if	Returns the number of elements satisfying a specific criteria.	`<hpx/algorithm.hpp>`	count_if[110]
hpx::equal	Determines if two sets of elements are the same.	`<hpx/algorithm.hpp>`	equal[111]
hpx::find	Finds the first element equal to a given value.	`<hpx/algorithm.hpp>`	find[112]
hpx::find_end	Finds the last sequence of elements in a certain range.	`<hpx/algorithm.hpp>`	find_end[113]
hpx::find_first_of	Searches for any one of a set of elements.	`<hpx/algorithm.hpp>`	find_first_of[114]
hpx::find_if	Finds the first element satisfying a specific criteria.	`<hpx/algorithm.hpp>`	find_if[115]
hpx::find_if_not	Finds the first element not satisfying a specific criteria.	`<hpx/algorithm.hpp>`	find_if_not[116]
hpx::for_each	Applies a function to a range of elements.	`<hpx/algorithm.hpp>`	for_each[117]
hpx::for_each_n	Applies a function to a number of elements.	`<hpx/algorithm.hpp>`	for_each_n[118]
hpx::parallel::v1::lexicographical_compare	Checks if a range of values is lexicographically less than another range of values.	`<hpx/algorithm.hpp>`	lexicographical_compare[119]
`hpx::parallel::v1::mismatch`	Finds the first position where two ranges differ.	`<hpx/algorithm.hpp>`	mismatch[120]
hpx::none_of	Checks if a predicate is `true` for none of the elements in a range.	`<hpx/algorithm.hpp>`	all_any_none_of[121]
hpx::parallel::v1::search	Searches for a range of elements.	`<hpx/algorithm.hpp>`	search[122]
hpx::parallel::v1::search_n	Searches for a number consecutive copies of an element in a range.	`<hpx/algorithm.hpp>`	search_n[123]

[106] http://en.cppreference.com/w/cpp/algorithm/adjacent_find
[107] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[108] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[109] http://en.cppreference.com/w/cpp/algorithm/count
[110] http://en.cppreference.com/w/cpp/algorithm/count_if
[111] http://en.cppreference.com/w/cpp/algorithm/equal
[112] http://en.cppreference.com/w/cpp/algorithm/find
[113] http://en.cppreference.com/w/cpp/algorithm/find_end
[114] http://en.cppreference.com/w/cpp/algorithm/find_first_of
[115] http://en.cppreference.com/w/cpp/algorithm/find_if
[116] http://en.cppreference.com/w/cpp/algorithm/find_if_not
[117] http://en.cppreference.com/w/cpp/algorithm/for_each
[118] http://en.cppreference.com/w/cpp/algorithm/for_each_n
[119] http://en.cppreference.com/w/cpp/algorithm/lexicographical_compare
[120] http://en.cppreference.com/w/cpp/algorithm/mismatch
[121] http://en.cppreference.com/w/cpp/algorithm/all_any_none_of
[122] http://en.cppreference.com/w/cpp/algorithm/search
[123] http://en.cppreference.com/w/cpp/algorithm/search_n

Table 2.16: Modifying parallel algorithms (In Header: <*hpx/algorithm.hpp*>)

Name	Description	In header	Algorithm page at cppreference.com
hpx::copy	Copies a range of elements to a new location.	<hpx/algorithm.hpp>	exclusive_scan[124]
hpx::copy_n	Copies a number of elements to a new location.	<hpx/algorithm.hpp>	copy_n[125]
hpx::copy_if	Copies the elements from a range to a new location for which the given predicate is `true`	<hpx/algorithm.hpp>	copy[126]
hpx::move	Moves a range of elements to a new location.	<hpx/algorithm.hpp>	move[127]
hpx::fill	Assigns a range of elements a certain value.	<hpx/algorithm.hpp>	fill[128]
hpx::fill_n	Assigns a value to a number of elements.	<hpx/algorithm.hpp>	fill_n[129]
hpx::generate	Saves the result of a function in a range.	<hpx/algorithm.hpp>	generate[130]
hpx::generate_n	Saves the result of N applications of a function.	<hpx/algorithm.hpp>	generate_n[131]
hpx::parallel::v1::remove	Removes the elements from a range that are equal to the given value.	<hpx/algorithm.hpp>	remove[132]
hpx::parallel::v1::remove_if	Removes the elements from a range that are equal to the given predicate is `false`	<hpx/algorithm.hpp>	remove[133]
hpx::parallel::v1::remove_copy	Copies the elements from a range to a new location that are not equal to the given value.	<hpx/algorithm.hpp>	remove_copy[134]
hpx::parallel::v1::remove_copy_if	Copies the elements from a range to a new location for which the given predicate is `false`	<hpx/algorithm.hpp>	remove_copy[135]
hpx::parallel::v1::replace	Replaces all values satisfying specific criteria with another value.	<hpx/algorithm.hpp>	replace[136]
hpx::parallel::v1::replace_if	Replaces all values satisfying specific criteria with another value.	<hpx/algorithm.hpp>	replace[137]
hpx::parallel::v1::replace_copy	Copies a range, replacing elements satisfying specific criteria with another value.	<hpx/algorithm.hpp>	replace_copy[138]
hpx::parallel::v1::replace_copy_if	Copies a range, replacing elements satisfying specific criteria with another value.	<hpx/algorithm.hpp>	replace_copy[139]
hpx::parallel::v1::reverse	Reverses the order elements in a range.	<hpx/algorithm.hpp>	reverse[140]
hpx::parallel::v1::reverse_copy	Creates a copy of a range that is reversed.	<hpx/algorithm.hpp>	reverse_copy[141]

Table 2.17: Set operations on sorted sequences (In Header: *<hpx/algorithm.hpp>*)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::merge	Merges two sorted ranges.	<hpx/algorithm.hpp>	merge[148]
hpx::parallel::v1::inplace_merge	Merges two ordered ranges in-place.	<hpx/algorithm.hpp>	inplace_merge[149]
hpx::parallel::v1::includes	Returns true if one set is a subset of another.	<hpx/algorithm.hpp>	includes[150]
hpx::parallel::v1::set_difference	Computes the difference between two sets.	<hpx/algorithm.hpp>	set_difference[151]
hpx::parallel::v1::set_intersection	Computes the intersection of two sets.	<hpx/algorithm.hpp>	set_intersection[152]
hpx::parallel::v1::set_symmetric_difference	Computes the symmetric difference between two sets.	<hpx/algorithm.hpp>	set_symmetric_difference[153]
hpx::parallel::v1::set_union	Computes the union of two sets.	<hpx/algorithm.hpp>	set_union[154]

[124] http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
[125] http://en.cppreference.com/w/cpp/algorithm/copy_n
[126] http://en.cppreference.com/w/cpp/algorithm/copy
[127] http://en.cppreference.com/w/cpp/algorithm/move
[128] http://en.cppreference.com/w/cpp/algorithm/fill
[129] http://en.cppreference.com/w/cpp/algorithm/fill_n
[130] http://en.cppreference.com/w/cpp/algorithm/generate
[131] http://en.cppreference.com/w/cpp/algorithm/generate_n
[132] http://en.cppreference.com/w/cpp/algorithm/remove
[133] http://en.cppreference.com/w/cpp/algorithm/remove
[134] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[135] http://en.cppreference.com/w/cpp/algorithm/remove_copy
[136] http://en.cppreference.com/w/cpp/algorithm/replace
[137] http://en.cppreference.com/w/cpp/algorithm/replace
[138] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[139] http://en.cppreference.com/w/cpp/algorithm/replace_copy
[140] http://en.cppreference.com/w/cpp/algorithm/reverse
[141] http://en.cppreference.com/w/cpp/algorithm/reverse_copy
[142] http://en.cppreference.com/w/cpp/algorithm/rotate
[143] http://en.cppreference.com/w/cpp/algorithm/rotate_copy
[144] http://en.cppreference.com/w/cpp/algorithm/swap_ranges
[145] http://en.cppreference.com/w/cpp/algorithm/transform
[146] http://en.cppreference.com/w/cpp/algorithm/unique
[147] http://en.cppreference.com/w/cpp/algorithm/unique_copy
[148] http://en.cppreference.com/w/cpp/algorithm/merge
[149] http://en.cppreference.com/w/cpp/algorithm/inplace_merge
[150] http://en.cppreference.com/w/cpp/algorithm/includes
[151] http://en.cppreference.com/w/cpp/algorithm/set_difference
[152] http://en.cppreference.com/w/cpp/algorithm/set_intersection
[153] http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
[154] http://en.cppreference.com/w/cpp/algorithm/set_union

Table 2.18: Heap operations (In Header: <hpx/algorithm.hpp>)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::is_heap	Returns `true` if the range is max heap.	`<hpx/algorithm.hpp>`	is_heap[155]
hpx::parallel::v1::is_heap_until	Returns the first element that breaks a max heap.	`<hpx/algorithm.hpp>`	is_heap_until[156]

Table 2.19: Minimum/maximum operations (In Header: <hpx/algorithm.hpp>)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::max_element	Returns the largest element in a range.	`<hpx/algorithm.hpp>`	max_element[157]
hpx::parallel::v1::min_element	Returns the smallest element in a range.	`<hpx/algorithm.hpp>`	min_element[158]
hpx::parallel::v1::minmax_element	Returns the smallest and the largest element in a range.	`<hpx/algorithm.hpp>`	minmax_element[159]

Table 2.20: Partitioning Operations (In Header: *<hpx/algorithm.hpp>*)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::is_partitioned	Returns `true` if each true element for a predicate precedes the false elements in a range.	`<hpx/algorithm.hpp>`	is_partitioned[160]
hpx::parallel::v1::partition	Divides elements into two groups without preserving their relative order.	`<hpx/algorithm.hpp>`	partition[161]
hpx::parallel::v1::partition_copy	Copies a range dividing the elements into two groups.	`<hpx/algorithm.hpp>`	partition_copy[162]
hpx::parallel::v1::stable_partition	Divides elements into two groups while preserving their relative order.	`<hpx/algorithm.hpp>`	stable_partition[163]

155 http://en.cppreference.com/w/cpp/algorithm/is_heap
156 http://en.cppreference.com/w/cpp/algorithm/is_heap_until
157 http://en.cppreference.com/w/cpp/algorithm/max_element
158 http://en.cppreference.com/w/cpp/algorithm/min_element
159 http://en.cppreference.com/w/cpp/algorithm/minmax_element
160 http://en.cppreference.com/w/cpp/algorithm/is_partitioned
161 http://en.cppreference.com/w/cpp/algorithm/partition
162 http://en.cppreference.com/w/cpp/algorithm/partition_copy
163 http://en.cppreference.com/w/cpp/algorithm/stable_partition

Table 2.21: Sorting Operations (In Header: <*hpx/algorithm.hpp*>)

Name	Description	In header	Algorithm page at cppreference.com
hpx::parallel::v1::is_sorted	Returns `true` if each element in a range is sorted.	`<hpx/algorithm.hpp>`	is_sorted[164]
hpx::parallel::v1::is_sorted_until	Returns the first unsorted element.	`<hpx/algorithm.hpp>`	is_sorted_until[165]
hpx::parallel::v1::sort	Sorts the elements in a range.	`<hpx/algorithm.hpp>`	sort[166]
hpx::parallel::v1::stable_sort	Sorts the elements in a range, maintain sequence of equal elements.	`<hpx/algorithm.hpp>`	stable_sort[167]
hpx::parallel::v1::sort_by_key	Sorts one range of data using keys supplied in another range.	`<hpx/algorithm.hpp>`	

[164] http://en.cppreference.com/w/cpp/algorithm/is_sorted
[165] http://en.cppreference.com/w/cpp/algorithm/is_sorted_until
[166] http://en.cppreference.com/w/cpp/algorithm/sort
[167] http://en.cppreference.com/w/cpp/algorithm/stable_sort

Table 2.22: Numeric Parallel Algorithms (In Header: <*hpx/numeric.hpp*>)

Name	Description	In header	Algorithm page at cppreference.com
`hpx::parallel::adjacent_difference`	Calculates the difference between each element in an input range and the preceding element.	`<hpx/numeric.hpp>`	adjacent_difference[168]
`hpx::parallel::exclusive_scan`	Does an exclusive parallel scan over a range of elements.	`<hpx/numeric.hpp>`	exclusive_scan[169]
`hpx::reduce`	Sums up a range of elements.	`<hpx/numeric.hpp>`	reduce[170]
`hpx::parallel::inclusive_scan`	Does an inclusive parallel scan over a range of elements.	`<hpx/algorithm.hpp>`	inclusive_scan[171]
`hpx::parallel::reduce_by_key`	Performs an inclusive scan on consecutive elements with matching keys, with a reduction to output only the final sum for each key. The key sequence `{1,1,1,2,3,3,3,3,1}` and value sequence `{2,3,4,5,6,7,8,9,10}` would be reduced to `keys={1,2,3,1}`, `values={9,5,30,10}`.	`<hpx/numeric.hpp>`	
`hpx::transform_reduce`	Sums up a range of elements after applying a function. Also, accumulates the inner products of two input ranges.	`<hpx/numeric.hpp>`	transform_reduce[172]
`hpx::parallel::transform_inclusive_scan`	Does an inclusive parallel scan over a range of elements after applying a function.	`<hpx/numeric.hpp>`	transform_inclusive_scan[173]
`hpx::parallel::transform_exclusive_scan`	Does an exclusive parallel scan over a range of elements after applying a function.	`<hpx/numeric.hpp>`	transform_exclusive_scan[174]

[168] http://en.cppreference.com/w/cpp/algorithm/adjacent_difference
[169] http://en.cppreference.com/w/cpp/algorithm/exclusive_scan
[170] http://en.cppreference.com/w/cpp/algorithm/reduce
[171] http://en.cppreference.com/w/cpp/algorithm/inclusive_scan
[172] http://en.cppreference.com/w/cpp/algorithm/transform_reduce
[173] http://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan
[174] http://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan

Table 2.23: Dynamic Memory Management (In Header: *<hpx/memory.hpp>*)

Name	Description	In header	Algorithm page at cppreference.com
hpx::destroy	Destroys a range of objects.	<hpx/memory.hpp>	destroy[175]
hpx::destroy_n	Destroys a range of objects.	<hpx/memory.hpp>	destroy_n[176]
hpx::parallel::v1::uninitialized_copy	Copies a range of objects to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_copy[177]
hpx::parallel::v1::uninitialized_copy_n	Copies a number of objects to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_copy_n[178]
hpx::parallel::v1::uninitialized_default_construct	Copies a range of objects to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_default_construct[179]
hpx::parallel::v1::uninitialized_default_construct_n	Copies a number of objects to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_default_construct_n[180]
hpx::parallel::v1::uninitialized_fill	Copies an object to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_fill[181]
hpx::parallel::v1::uninitialized_fill_n	Copies an object to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_fill_n[182]
hpx::parallel::v1::uninitialized_move	Moves a range of objects to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_move[183]
hpx::parallel::v1::uninitialized_move_n	Moves a number of objects to an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_move_n[184]
hpx::parallel::v1::uninitialized_value_construct	Constructs objects in an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_value_construct[185]
hpx::parallel::v1::uninitialized_value_construct_n	Constructs objects in an uninitialized area of memory.	<hpx/memory.hpp>	uninitialized_value_construct_n[186]

175 http://en.cppreference.com/w/cpp/memory/destroy
176 http://en.cppreference.com/w/cpp/memory/destroy_n
177 http://en.cppreference.com/w/cpp/memory/uninitialized_copy
178 http://en.cppreference.com/w/cpp/memory/uninitialized_copy_n
179 http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct
180 http://en.cppreference.com/w/cpp/memory/uninitialized_default_construct_n
181 http://en.cppreference.com/w/cpp/memory/uninitialized_fill
182 http://en.cppreference.com/w/cpp/memory/uninitialized_fill_n
183 http://en.cppreference.com/w/cpp/memory/uninitialized_move
184 http://en.cppreference.com/w/cpp/memory/uninitialized_move_n
185 http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct
186 http://en.cppreference.com/w/cpp/memory/uninitialized_value_construct_n

Table 2.24: Index-based for-loops (In Header: <*hpx/algorithm.hpp*>)

Name	Description	In header
hpx::for_loop	Implements loop functionality over a range specified by integral or iterator bounds.	`<hpx/algorithm.hpp>`
hpx::for_loop_strided	Implements loop functionality over a range specified by integral or iterator bounds.	`<hpx/algorithm.hpp>`
hpx::for_loop_n	Implements loop functionality over a range specified by integral or iterator bounds.	`<hpx/algorithm.hpp>`
hpx::for_loop_n_strided	Implements loop functionality over a range specified by integral or iterator bounds.	`<hpx/algorithm.hpp>`

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:

```
std::size_t chunk_size =
    executor_parameter_traits<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::parallel::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::parallel::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::parallel::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::parallel::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::parallel::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::parallel::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::parallel::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[187] are based on the `task_block` concept that is a part of the common subset of the Microsoft Parallel Patterns Library (PPL)[188] and the Intel Threading Building Blocks (TBB)[189] 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++[190] and async and finish from X10[191]. They improve on existing practice in the following ways:

- The exception handling model is simplified and more consistent with normal C++ exceptions.
- Most violations of strict fork-join parallelism can be enforced at compile time (with compiler assistance, in some cases).
- The syntax allows scheduling approaches other than child stealing.

Consider an example of a parallel traversal of a tree, where a user-provided function compute is applied to each node of the tree, returning the sum of the results:

```
template <typename Func>
int traverse(node& n, Func && compute)
{
    int left = 0, right = 0;
    define_task_block(
        [&](task_block<>& tr) {
            if (n.left)
                tr.run([&] { left = traverse(*n.left, compute); });
            if (n.right)
                tr.run([&] { right = traverse(*n.right, compute); });
        });

    return compute(n) + left + right;
}
```

The example above demonstrates the use of two of the functions, `hpx::parallel::define_task_block` and the `hpx::parallel::task_block::run` member function of a `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

187 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
188 https://msdn.microsoft.com/en-us/library/dd492418.aspx
189 https://www.threadingbuildingblocks.org/
190 https://software.intel.com/en-us/articles/intel-cilk-plus/
191 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`:

```
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[192]. The main addition is that a `task_block` can be invoked with an execution policy as its first argument, very similar to the parallel algorithms.

An execution policy is an object that expresses the requirements on the ordering of functions invoked as a consequence of the invocation of a task block. Enabling passing an execution policy to `define_task_block` gives the user control over the amount of parallelism employed by the created `task_block`. In the following example the use of an explicit `par` execution policy makes the user's intent explicit:

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

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

(continues on next page)

[192] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf

(continued from previous page)

```
    return compute(n) + left + right;
}
```

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::parallel::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::parallel::execution::parallel_policy*.

HPX also adds the ability to access the execution policy that was used to create a given `task_block`.

Using executors to run tasks

Often, users want to be able to not only define an execution policy to use by default for all spawned tasks inside the task block, but also to customize the execution context for one of the tasks executed by `task_block::run`. Adding an optionally passed executor instance to that function enables this use case:

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

    define_task_block(
        execution::par,                 // execution::parallel_policy
        [&](auto& tb) {
            if (n->left)
            {
                // use explicitly specified executor to run this task
                tb.run(my_executor(), [&] { left = traverse(n->left, compute); });
            }
            if (n->right)
            {
                // use the executor associated with the par execution policy
                tb.run([&] { right = traverse(n->right, compute); });
            }
        });

    return compute(n) + left + right;
}
```

HPX still supports calling *hpx::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*.

2.5.7 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 *HPX_PLAIN_ACTION* can be used to define the action type. Here is an example demonstrating this:

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }
}

// This will define the action type 'some_global_action' which represents
// the function 'app::some_global_function'.
HPX_PLAIN_ACTION(app::some_global_function, some_global_action);
```

Important: The macro *HPX_PLAIN_ACTION* has to be placed in global namespace, even if the wrapped function is located in some other namespace. The newly defined action type is placed in the global namespace as well.

If the action type should be defined somewhere not in global namespace, the action type definition has to be split into two macro invocations (*HPX_DEFINE_PLAIN_ACTION* and *HPX_REGISTER_ACTION*) as shown in the next example:

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }

    // On conforming compilers the following macro expands to:
    //
    //    typedef hpx::actions::make_action<
    //        decltype(&some_global_function), &some_global_function
    //    >::type some_global_action;
    //
    // This will define the action type 'some_global_action' which represents
    // the function 'some_global_function'.
    HPX_DEFINE_PLAIN_ACTION(some_global_function, some_global_action);
}

// The following macro expands to a series of definitions of global objects
// which are needed for proper serialization and initialization support
// enabling the remote invocation of the function``some_global_function``
HPX_REGISTER_ACTION(app::some_global_action, app_some_global_action);
```

The shown code defines an action type `some_global_action` inside the namespace `app`.

Important: If the action type definition is split between two macros as shown above, the name of the action type to create has to be the same for both macro invocations (here `some_global_action`).

Important: The second argument passed to *HPX_REGISTER_ACTION* (app_some_global_action) has to comprise a globally unique C++ identifier representing the action. This is used for serialization purposes.

For member functions of objects which have been registered with *AGAS* (e.g. 'components') a different registration macro *HPX_DEFINE_COMPONENT_ACTION* has to be utilized. Any component needs to be declared in a header file and have some special support macros defined in a source file. Here is an example demonstrating this. The first snippet has to go into the header file:

```
namespace app
{
    struct some_component
      : hpx::components::component_base<some_component>
    {
        int some_member_function(std::string s)
        {
            return boost::lexical_cast<int>(s);
        }

        // This will define the action type 'some_member_action' which
        // represents the member function 'some_member_function' of the
        // object type 'some_component'.
        HPX_DEFINE_COMPONENT_ACTION(some_component, some_member_function,
            some_member_action);
    };
}

// Note: The second argument to the macro below has to be systemwide-unique
//       C++ identifiers
HPX_REGISTER_ACTION_DECLARATION(app::some_component::some_member_action, some_
↪component_some_action);
```

The next snippet belongs into a source file (e.g. the main application source file) in the simplest case:

```
typedef hpx::components::component<app::some_component> component_type;
typedef app::some_component some_component;

HPX_REGISTER_COMPONENT(component_type, some_component);

// The parameters for this macro have to be the same as used in the corresponding
// HPX_REGISTER_ACTION_DECLARATION() macro invocation above
typedef some_component::some_member_action some_component_some_action;
HPX_REGISTER_ACTION(some_component_some_action);
```

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 *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.

R f(p...)	Synchronous Execution (returns R)	Asynchronous Execution (returns future<R>)	Fire & Forget Execution (returns **void**)
Functions (direct invocation)	f(p...) C++	async(f, p...)	apply(f, p...)
Functions (lazy invocation)	bind(f, p...)(...)	async(bind(f, p...), ...) C++ Standard Library	apply(bind(f, p...), ...)
Actions (direct invocation)	HPX_ACTION(f, action) a(id, p...)	HPX_ACTION(f, action) async(a, id, p...)	HPX_ACTION(f, action) apply(a, id, p...)
Actions (lazy invocation)	HPX_ACTION(f, action) bind(a, id, p...)(...)	HPX_ACTION(f, action) async(bind(a, id, p...), ...)	HPX_ACTION(f, action) apply(bind(a, id, p...), ...) HPX

Fig. 2.8: Overview of the main API exposed by *HPX*.

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.

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):

```
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:

```
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*:

```
some_global_action act;     // define an instance of some_global_action
hpx::future<void> f = hpx::async(act, hpx::find_here(), 2.0);
//
// ... other code can be executed here
//
f.get();    // this will possibly wait for the asynchronous operation to 'return'
```

(as before, the function `hpx::find_here()` returns the id of the local *locality* (the *locality* this code is executed on).

Note: The use of a `hpx::future<void>` allows the current thread to synchronize with any remote operation not returning any value.

Note: Any `std::future<>` returned from `std::async()` is required to block in its destructor if the value has not been set for this future yet. This is not true for `hpx::future<>` which will never block in its destructor, even if the value has not been returned to the future yet. We believe that consistency in the behavior of futures is more important than standards conformance in this case.

Any component member function can be invoked using the same syntactic construct. Given that `id` is the global address for a component instance created earlier, this invocation looks like:

```
some_component_action act;     // define an instance of some_component_action
hpx::future<int> f = hpx::async(act, id, "42");
//
// ... other code can be executed here
//
cout << f.get();    // this will possibly wait for the asynchronous operation to
↪'return' 42
```

Note: The invocation of `f.get()` will return the result immediately (without suspending the calling thread) if the result from the asynchronous operation has already been returned. Otherwise, the invocation of `f.get()` will suspend the execution of the calling thread until the asynchronous operation returns its result.

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:

```
some_global_action act;     // define an instance of some_global_action
act(hpx::find_here(), 2.0);
```

While this call looks just like a normal synchronous function invocation, the function wrapped by the action will be scheduled to run on a new thread and the calling thread will be suspended. After the new thread has executed the wrapped global function, the waiting thread will resume and return from the synchronous call.

Equivalently, any action wrapping a component member function can be invoked synchronously as follows:

```
some_component_action act;     // define an instance of some_component_action
int result = act(id, "42");
```

The action invocation will either schedule a new thread locally to execute the wrapped member function (as before, `id` is the global address of the component instance the member function should be invoked on), or it will send a parcel to the remote *locality* of the component causing a new thread to be scheduled there. The calling thread will be suspended until the function returns its result. This result will be returned from the synchronous action invocation.

It is very important to understand that this 'synchronous' invocation syntax in fact conceals an asynchronous function call. This is beneficial as the calling thread is suspended while waiting for the outcome of a potentially remote operation. The *HPX* thread scheduler will schedule other work in the meantime, allowing the application to make further progress while the remote result is computed. This helps overlapping computation with communication and hiding communication latencies.

Note: The syntax of 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:

```
// first action
std::int32_t action1(std::int32_t i)
{
    return i+1;
}
HPX_PLAIN_ACTION(action1);    // defines action1_type

// second action
std::int32_t action2(std::int32_t i)
{
    return i*2;
}
HPX_PLAIN_ACTION(action2);    // defines action2_type

// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;     // define an instance of 'action1_type'
action2_type act2;     // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1, hpx::make_continuation(act2),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n";   // will print: 86 ((42 + 1) * 2)
```

By default, the continuation is executed on the same *locality* as `hpx::async_continue` is invoked from. If you want to specify the *locality* where the continuation should be executed, the code above has to be written as:

```
// this code invokes 'action1' above and passes along a continuation
// function which will forward the result returned from 'action1' to
// 'action2'.
action1_type act1;     // define an instance of 'action1_type'
action2_type act2;     // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1, hpx::make_continuation(act2, hpx::find_here()),
```

(continues on next page)

(continued from previous page)

```
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n";   // will print: 86 ((42 + 1) * 2)
```

Similarly, it is possible to chain more than 2 operations:

```
action1_type act1;      // define an instance of 'action1_type'
action2_type act2;      // define an instance of 'action2_type'
hpx::future<int> f =
    hpx::async_continue(act1,
        hpx::make_continuation(act2, hpx::make_continuation(act1)),
        hpx::find_here(), 42);
hpx::cout << f.get() << "\n";   // will print: 87 ((42 + 1) * 2 + 1)
```

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

```
struct continuation
{
    template <typename Result>
    void operator()(hpx::id_type id, Result&& result) const
    {
        ...
    }
};
```

where the parameters passed to the overloaded function operator `operator()()` are:

- the `id` is the global id where the final result of the asynchronous chain of operations should be sent to (in most cases this is the id of the `hpx::future` returned from the initial call to `hpx::async_continue`. Any custom continuation function should make sure this `id` is forwarded to the last operation in the chain.
- the `result` is the result value of the current operation in the asynchronous execution chain. This value needs to be forwarded to the next operation.

Note: All of those operations are implemented by the predefined continuation function object which is returned from `hpx::make_continuation`. Any (custom) function object used as a continuation should conform to the same interface.

Action error handling

Like in any other asynchronous invocation scheme it is important to be able to handle error conditions occurring while the asynchronous (and possibly remote) operation is executed. In *HPX* all error handling is based on standard C++ exception handling. Any exception thrown during the execution of an asynchronous operation will be transferred back to the original invocation *locality*, where it is rethrown during synchronization with the calling thread.

Important: Exceptions thrown during asynchronous execution can be transferred back to the invoking thread only for the synchronous and the asynchronous case with synchronization. Like with any other unhandled exception, any exception thrown during the execution of an asynchronous action *without* synchronization will result in calling `hpx::terminate` causing the running application to exit immediately.

Note: Even if error handling internally relies on exceptions, most of the API functions exposed by *HPX* can be used without throwing an exception. Please see *Working with exceptions* for more information.

As an example, we will assume that the following remote function will be executed:

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

```
// note: wrapped function will throw hpx::exception
some_error_action act;            // define an instance of some_error_action
try {
    act(hpx::find_here(), -3);    // exception will be rethrown from here
}
catch (hpx::exception const& e) {
    // prints: 'some really bad error happened: HPX(bad parameter)'
    cout << e.what();
}
```

If this action is invoked asynchronously with synchronization, the exception is propagated to the waiting thread as well and is re-thrown from the future's function `get()`:

```
// note: wrapped function will throw hpx::exception
some_error_action act;            // define an instance of some_error_action
hpx::future<void> f = hpx::async(act, hpx::find_here(), -3);
try {
    f.get();                      // exception will be rethrown from here
}
catch (hpx::exception const& e) {
    // prints: 'some really bad error happened: HPX(bad parameter)'
    cout << e.what();
}
```

For more information about error handling please refer to the section *Working with exceptions*. There we also explain how to handle error conditions without having to rely on exception.

Writing components

A component in *HPX* is a C++ class which can be created remotely and for which its member functions can be invoked remotely as well. The following sections highlight how components can be defined, created, and used.

Defining components

In order for a C++ class type to be managed remotely in *HPX*, the type must be derived from the `hpx::components::component_base` template type. We call such C++ class types 'components'.

Note that the component type itself is passed as a template argument to the base class:

```
// header file some_component.hpp

#include <hpx/include/components.hpp>

namespace app
{
    // Define a new component type 'some_component'
    struct some_component
      : hpx::components::component_base<some_component>
    {
        // This member function is has to be invoked remotely
        int some_member_function(std::string const& s)
        {
            return boost::lexical_cast<int>(s);
        }

        // This will define the action type 'some_member_action' which
        // represents the member function 'some_member_function' of the
        // object type 'some_component'.
        HPX_DEFINE_COMPONENT_ACTION(some_component, some_member_function, some_member_
↪action);
    };
}

// This will generate the necessary boiler-plate code for the action allowing
// it to be invoked remotely. This declaration macro has to be placed in the
// header file defining the component itself.
//
// Note: The second argument to the macro below has to be systemwide-unique
//       C++ identifiers
//
HPX_REGISTER_ACTION_DECLARATION(app::some_component::some_member_action, some_
↪component_some_action);
```

There is more boiler plate code which has to be placed into a source file in order for the component to be usable. Every component type is required to have macros placed into its source file, one for each component type and one macro for each of the actions defined by the component type.

For instance:

```
// source file some_component.cpp

#include "some_component.hpp"

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

(continues on next page)

(continued from previous page)

```
// remote creation of 'app::some_component' instances with 'hpx::new_<>()'
//
using some_component = app::some_component;
using some_component_type = hpx::components::component<some_component>;

// Please note that the second argument to this macro must be a
// (system-wide) unique C++-style identifier (without any namespaces)
//
HPX_REGISTER_COMPONENT(some_component_type, some_component);

// The parameters for this macro have to be the same as used in the corresponding
// HPX_REGISTER_ACTION_DECLARATION() macro invocation in the corresponding
// header file.
//
// Please note that the second argument to this macro must be a
// (system-wide) unique C++-style identifier (without any namespaces)
//
HPX_REGISTER_ACTION(app::some_component::some_member_action, some_component_some_
↪action);
```

Defining client side representation classes

Often it is very convenient to define a separate type for a component which can be used on the client side (from where the component is instantiated and used). This step might seem as unnecessary duplicating code, however it significantly increases the type safety of the code.

A possible implementation of such a client side representation for the component described in the previous section could look like:

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

namespace app
{
    // Define a client side representation type for the component type
    // 'some_component' defined in the previous section.
    //
    struct some_component_client
      : hpx::components::client_base<some_component_client, some_component>
    {
        using base_type = hpx::components::client_base<
                some_component_client, some_component>;

        some_component_client(hpx::future<hpx::id_type> && id)
          : base_type(std::move(id))
        {}

        hpx::future<int> some_member_function(std::string const& s)
        {
            some_component::some_member_action act;
            return hpx::async(act, get_id(), s);
        }
    };
}
```

A client side object stores the global id of the component instance it represents. This global id is accessible by calling the function `client_base<>::get_id()`. The special constructor which is provided in the example allows to create this client side object directly using the API function `hpx::new_`.

Creating component instances

Instances of defined component types can be created in two different ways. If the component to create has a defined client side representation type, then this can be used, otherwise use the server type.

The following examples assume that `some_component_type` is the type of the server side implementation of the component to create. All additional arguments (see `, ...` notation below) are passed through to the corresponding constructor calls of those objects:

```
// create one instance on the given locality
hpx::id_type here = hpx::find_here();
hpx::future<hpx::id_type> f =
    hpx::new_<some_component_type>(here, ...);

// create one instance using the given distribution
// policy (here: hpx::colocating_distribution_policy)
hpx::id_type here = hpx::find_here();
hpx::future<hpx::id_type> f =
    hpx::new_<some_component_type>(hpx::colocated(here), ...);

// create multiple instances on the given locality
hpx::id_type here = find_here();
hpx::future<std::vector<hpx::id_type>> f =
    hpx::new_<some_component_type[]>(here, num, ...);

// create multiple instances using the given distribution
// policy (here: hpx::binpacking_distribution_policy)
hpx::future<std::vector<hpx::id_type>> f = hpx::new_<some_component_type[]>(
    hpx::binpacking(hpx::find_all_localities()), num, ...);
```

The examples below demonstrate the use of the same API functions for creating client side representation objects (instead of just plain ids). These examples assume that `client_type` is the type of the client side representation of the component type to create. As above, all additional arguments (see `, ...` notation below) are passed through to the corresponding constructor calls of the server side implementation objects corresponding to the `client_type`:

```
// create one instance on the given locality
hpx::id_type here = hpx::find_here();
client_type c = hpx::new_<client_type>(here, ...);

// create one instance using the given distribution
// policy (here: hpx::colocating_distribution_policy)
hpx::id_type here = hpx::find_here();
client_type c = hpx::new_<client_type>(hpx::colocated(here), ...);

// 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:

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

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

// By default, the number of segments is equal to the current number of
// localities
//
hpx::partitioned_vector<double> va(50);
hpx::partitioned_vector<double> vb(50, 0.0);
```

An example of how to use the `partitioned_vector` container with distribution policies would be:

```
#include <hpx/include/partitioned_vector.hpp>
#include <hpx/runtime/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)

(continued from previous page)

```
// The number of segments is 10 and those segments are spread across the
// localities collected in the variable locs in a Round-Robin manner
//
hpx::partitioned_vector<double> va(50, layout);
hpx::partitioned_vector<double> vb(50, 0.0, layout);
```

By definition, a segmented container must be accessible from any thread although its construction is synchronous only for the thread who has called its constructor. To overcome this problem, it is possible to assign a symbolic name to the segmented container:

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

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

hpx::future<void> fserver = hpx::async(
  [](){
    hpx::partitioned_vector<double> v(50);

    // Register the 'partitioned_vector' with the name "some_name"
    //
    v.register_as("some_name");

    /* Do some code  */
  });

hpx::future<void> fclient =
  hpx::async(
    [](){
      // Naked 'partitioned_vector'
      //
      hpx::partitioned_vector<double> v;

      // Now the variable v points to the same 'partitioned_vector' that has
      // been registered with the name "some_name"
      //
      v.connect_to("some_name");

      /* Do some code  */
    });
```

Segmented containers

HPX provides the following segmented containers:

Table 2.25: Sequence containers

Name	Description	In header	Class page at cppreference.com
`hpx::partitioned_vector`	Dynamic segmented contiguous array.	`<hpx/include/partitioned_vector.hpp>`	vector[193]

Table 2.26: Unordered associative containers

Name	Description	In header	Class page at cppreference.com
hpx::unordered_map	Segmented collection of key-value pairs, hashed by keys, keys are unique.	<hpx/include/unordered_map.hpp>	unordered_map[194]

Segmented iterators and segmented iterator traits

The basic iterator used in the STL library is only suitable for one-dimensional structures. The iterators we use in *HPX* must adapt to the segmented format of our containers. Our iterators are then able to know when incrementing themselves if the next element of type `T` is in the same data segment or in another segment. In this second case, the iterator will automatically point to the beginning of the next segment.

Note: Note that the dereference operation `operator *` does not directly return a reference of type `T&` but an intermediate object wrapping this reference. When this object is used as an l-value, a remote write operation is performed; When this object is used as an r-value, implicit conversion to `T` type will take care of performing remote read operation.

It is sometimes useful not only to iterate element by element, but also segment by segment, or simply get a local iterator in order to avoid additional construction costs at each deferencing operations. To mitigate this need, the `hpx::traits::segmented_iterator_traits` are used.

With `segmented_iterator_traits` users can uniformly get the iterators which specifically iterates over segments (by providing a segmented iterator as a parameter), or get the local begin/end iterators of the nearest local segment (by providing a per-segment iterator as a parameter):

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

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

using iterator = hpx::partitioned_vector<T>::iterator;
using traits   = hpx::traits::segmented_iterator_traits<iterator>;

hpx::partitioned_vector<T> v;
std::size_t count = 0;

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)
```

(continues on next page)

[193] http://en.cppreference.com/w/cpp/container/vector
[194] http://en.cppreference.com/w/cpp/container/unordered_map

(continued from previous page)

```
    {
        *lit = count;
    }
}
```

Which is equivalent to:

```
hpx::partitioned_vector<T> v;
std::size_t count = 0;

auto begin = v.begin();
auto end   = v.end();

for (auto it = begin; it != end; ++it, ++count)
{
    *it = count;
}
```

Using views

The use of multidimensional arrays is quite common in the numerical field whether to perform dense matrix operations or to process images. It exist many libraries which implement such object classes overloading their basic operators (e.g.``+``, -, *, (), etc.). However, such operation becomes more delicate when the underlying data layout is segmented or when it is mandatory to use optimized linear algebra subroutines (i.e. BLAS subroutines).

Our solution is thus to relax the level of abstraction by allowing the user to work not directly on n-dimensionnal data, but on "n-dimensionnal collections of 1-D arrays". The use of well-accepted techniques on contiguous data is thus preserved at the segment level, and the composability of the segments is made possible thanks to multidimensional array-inspired access mode.

Preface: Why SPMD?

Although *HPX* refutes by design this programming model, the *locality* plays a dominant role when it comes to implement vectorized code. To maximize local computations and avoid unneeded data transfers, a parallel section (or Single Programming Multiple Data section) is required. Because the use of global variables is prohibited, this parallel section is created via the RAII idiom.

To define a parallel section, simply write an action taking a `spmd_block` variable as a first parameter:

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

void bulk_function(hpx::lcos::spmd_block block /* , arg0, arg1, ... */)
{
    // Parallel section

    /* Do some code */
}
HPX_PLAIN_ACTION(bulk_function, bulk_action);
```

Note: In the following paragraphs, we will use the term "image" several times. An image is defined as a lightweight process whose entry point is a function provided by the user. It's an "image of the function".

The `spmd_block` class contains the following methods:

- [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:

```
#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 `vv(i,j)` which would call internally the expression `v[k]`.

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

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

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

using iterator = hpx::partitioned_vector<double>::iterator;
using traits   = hpx::traits::segmented_iterator_traits<iterator>;

hpx::partitioned_vector<double> v;

// Create a 1-D view of the vector of segments
auto vv = traits::segment(v.begin());

// Access segment i
std::vector<double> v = vv[i];
```

Our views are called "multidimensional" in the sense that they generalize to N dimensions the purpose of `segmented_iterator_traits::segment()` in the 1-D case. Note that in a parallel section, the 2-D expression `a(i,j) = b(i,j)` is quite confusing because without convention, each of the images invoked will race to execute the statement. For this reason, our views are not only multidimensional but also “spmd-aware”.

Note: SPMD-awareness: The convention is simple. If an assignment statement contains a view subscript as an l-value, it is only and only the image holding the r-value who is evaluating the statement. (In MPI sense, it is called a Put operation).

Subscript-based operations

Here are some examples of using subscripts in the 2-D view case:

```
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(double);

using Vec = hpx::partitioned_vector<double>;
using View_2D = hpx::partitioned_vector_view<double,2>;

/* Do some code */

Vec v;

// Parallel section (suppose 'block' an spmd_block instance)
{
    std::size_t height, width;

    // Instantiate the view
    View_2D vv(block, v.begin(), v.end(), {height,width});

    // The l-value is a view subscript, the image that owns vv(1,0)
    // evaluates the assignment.
    vv(0,1) = vv(1,0);

    // The l-value is a view subscript, the image that owns the r-value
    // (result of expression 'std::vector<double>(4,1.0)') evaluates the
    // assignment : oops! race between all participating images.
    vv(2,3) = std::vector<double>(4,1.0);
}
```

Iterator-based operations

Here are some examples of using iterators in the 3-D view case:

```
#include <hpx/components/containers/partitioned_vector/partitioned_vector_view.hpp>
#include <hpx/include/partitioned_vector.hpp>

// The following code generates all necessary boiler plate to enable the
// remote creation of 'partitioned_vector' segments
//
HPX_REGISTER_PARTITIONED_VECTOR(int);
```

(continues on next page)

(continued from previous 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:

```
#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>;
```

(continues on next page)

(continued from previous page)

```
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(
```

(continues on next page)

(continued from previous page)

```
        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:

```
#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 2.27: Parameters of coarray constructor

Parameter	Description
`block`	Reference to a `spmd_block` object
`"a"`	Symbolic name of type `std::string`
`{height,width,_}`	Dimensions of the `coarray` object
`segment_size`	Size of a co-indexed element (i.e. size of the object referenced by the expression `a(i,j,k)`)

Note that the "last dimension size" cannot be set by the user. It only accepts the constexpr variable `hpx::container::placeholders::_`. This size, which is considered private, is equal to the number of current images (value returned by `block.get_num_images()`).

Note: An important constraint to remember about coarray objects is that all segments sharing the same "last dimension index" are located in the same image.

Using co-arrays

The member functions owned by the `coarray` objects are exactly the same as those of spmd multidimensional views. These are:

```
* Subscript-based operations
* Iterator-based operations
```

However, one additional functionality is provided. Knowing that the element `a(i,j,k)` is in the memory of the `k`th image, the use of local subscripts is possible.

Note: For spmd multidimensional views, subscripts are only global as it still involves potential remote data transfers.

Here is an example of using local subscripts:

```
#include <hpx/components/containers/coarray/coarray.hpp>
#include <hpx/collectives/spmd_block.hpp>

// The following code generates all necessary boiler plate to enable the
// co-creation of 'coarray'
//
HPX_REGISTER_COARRAY(double);

// Parallel section (suppose 'block' an spmd_block instance)
{
    using hpx::container::placeholders::_;

    std::size_t height=32, width=4, segment_size=10;

    hpx::coarray<double,3> a(block, "a", {height,width,_}, segment_size);

    double idx = block.this_image()*height*width;

    for (std::size_t j = 0; j<width; j++)
    for (std::size_t i = 0; i<height; i++)
```

(continues on next page)

(continued from previous 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()`.

2.5.8 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[195] 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: Hello world*.

```
#!/bin/bash
#
#PBS -l nodes=2:ppn=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

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

Caution: If the first application specific argument (inside `$APP_OPTIONS`) is a non-option (i.e., does not start with a `-` or a `--`), then the argument has to be placed before the option *--hpx:nodes*, which, in this case, should be the last option on the command line.

Alternatively, use the option *--hpx:endnodes* to explicitly mark the end of the list of node names:

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

[195] http://www.clusterresources.com/torquedocs21/

The `#PBS -l nodes=2:ppn=4` directive will cause two compute nodes to be allocated for the application, as specified in the option `nodes`. Each of the nodes will dedicate four cores to the program, as per the option `ppn`, short for "processors per node" (PBS does not distinguish between processors and cores). Note that requesting more cores per node than physically available is pointless and may prevent PBS from accepting the script.

On newer PBS versions the PBS command syntax might be different. For instance, the PBS script above would look like:

```
#!/bin/bash
#
#PBS -l select=2:ncpus=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

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

`APP_PATH` and `APP_OPTIONS` are shell variables that respectively specify the correct path to the executable (`hello_world_distributed` in this case) and the command line options. Since the `hello_world_distributed` application doesn't need any command line options, `APP_OPTIONS` has been left empty. Unlike in other execution environments, there is no need to use the *`--hpx:threads`* option to indicate the required number of OS threads per node; the *HPX* library will derive this parameter automatically from PBS.

Finally, pbsdsh is a PBS command that starts tasks to the resources allocated to the current job. It is recommended to leave this line as shown and modify only the PBS options and shell variables as needed for a specific application.

Important: A script invoked by pbsdsh starts in a very basic environment: the user's `$HOME` directory is defined and is the current directory, the `LANG` variable is set to `C` and the `PATH` is set to the basic `/usr/local/bin:/usr/bin:/bin` as defined in a system-wide file pbs_environment. Nothing that would normally be set up by a system shell profile or user shell profile is defined, unlike the environment for the main job script.

Another choice is for the pbsdsh command in your main job script to invoke your program via a shell, like `sh` or `bash`, so that it gives an initialized environment for each instance. Users can create a small script `runme.sh`, which is used to invoke the program:

```
#!/bin/bash
# Small script which invokes the program based on what was passed on its
# command line.
#
# This script is executed by the bash shell which will initialize all
# environment variables as usual.
$@
```

Now, the script is invoked using the pbsdsh tool:

```
#!/bin/bash
#
#PBS -l nodes=2:ppn=4

APP_PATH=~/packages/hpx/bin/hello_world_distributed
APP_OPTIONS=

pbsdsh -u runme.sh $APP_PATH $APP_OPTIONS --hpx:nodes=`cat $PBS_NODEFILE`
```

All that remains now is submitting the job to the queuing system. Assuming that the contents of the PBS script were saved in the file `pbs_hello_world.sh` in the current directory, this is accomplished by typing:

```
qsub ./pbs_hello_world_pbs.sh
```

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

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

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

```
qstat 42.supercomputer.some.university.edu
```

and look for a single-letter job status symbol. The common cases include:

- *Q* - signifies that the job is queued and awaiting its turn to be executed.
- *R* - indicates that the job is currently running.
- *C* - means that the job has completed.

The example qstat output below shows a job waiting for execution resources to become available:

```
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
42.supercomputer          ...ello_world.sh joe_user               0 Q batch
```

After the job completes, PBS will place two files, `pbs_hello_world.sh.o42` and `pbs_hello_world.sh.e42`, in the directory where the job was submitted. The first contains the standard output and the second contains the standard error from all the nodes on which the application executed. In our example, the error output file should be empty and the standard output file should contain something similar to:

```
hello world from OS-thread 3 on locality 0
hello world from OS-thread 2 on locality 0
hello world from OS-thread 1 on locality 1
hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 1
hello world from OS-thread 2 on locality 1
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 1
```

Congratulations! You have just run your first distributed *HPX* application!

How to use *HPX* applications with SLURM

Just like PBS (described in section *How to use HPX applications with PBS*), SLURM is a job management system which is widely used on large supercomputing systems. Any *HPX* application can easily be run using SLURM. This section describes how this can be done.

The easiest way to run an *HPX* application using SLURM is to utilize the command line tool srun, which interacts with the SLURM batch scheduling system:

```
srun -p <partition> -N <number-of-nodes> hpx-application <application-arguments>
```

Here, `<partition>` is one of the node partitions existing on the target machine (consult the machine's documentation to get a list of existing partitions) and `<number-of-nodes>` is the number of compute nodes that should be used. By default, the *HPX* application is started with one *locality* per node and uses all available cores on a node. You can change the number of localities started per node (for example, to account for NUMA effects) by specifying the `-n` option of srun. The number of cores per *locality* can be set by `-c`. The `<application-arguments>` are any application specific arguments that need to be passed on to the application.

Note: There is no need to use any of the *HPX* command line options related to the number of localities, number of threads, or related to networking ports. All of this information is automatically extracted from the SLURM environment by the *HPX* startup code.

Important: The srun documentation explicitly states: "If `-c` is specified without `-n`, as many tasks will be allocated per node as possible while satisfying the `-c` restriction. For instance on a cluster with 8 CPUs per node, a job request for 4 nodes and 3 CPUs per task may be allocated 3 or 6 CPUs per node (1 or 2 tasks per node) depending upon resource consumption by other jobs." For this reason, it's recommended to always specify `-n <number-of-instances>`, even if `<number-of-instances>` is equal to one (`1`).

Interactive shells

To get an interactive development shell on one of the nodes, users can issue the following command:

```
srun -p <node-type> -N <number-of-nodes> --pty /bin/bash -l
```

After the shell has been opened, users can run their *HPX* application. By default, it uses all available cores. Note that if you requested one node, you don't need to do `srun` again. However, if you requested more than one node, and want to run your distributed application, you can use `srun` again to start up the distributed *HPX* application. It will use the resources that have been requested for the interactive shell.

Scheduling batch jobs

The above mentioned method of running *HPX* applications is fine for development purposes. The disadvantage that comes with `srun` is that it only returns once the application is finished. This might not be appropriate for longer-running applications (for example, benchmarks or larger scale simulations). In order to cope with that limitation, users can use the sbatch command.

The `sbatch` command expects a script that it can run once the requested resources are available. In order to request resources, users need to add `#SBATCH` comments in their script or provide the necessary parameters to `sbatch` directly. The parameters are the same as with `run`. The commands you need to execute are the same you would need to start your application as if you were in an interactive shell.

2.5.9 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.

2.5.10 Optimizing *HPX* applications

Performance counters

Performance counters in *HPX* are used to provide information as to how well the runtime system or an application is performing. The counter data can help determine system bottlenecks, and fine-tune system and application performance. The *HPX* runtime system, its networking, and other layers provide counter data that an application can consume to provide users with information about how well the application is performing.

Applications can also use counter data to determine how much system resources to consume. For example, an application that transfers data over the network could consume counter data from a network switch to determine how much data to transfer without competing for network bandwidth with other network traffic. The application could use the counter data to adjust its transfer rate as the bandwidth usage from other network traffic increases or decreases.

Performance counters are *HPX* parallel processes that expose a predefined interface. *HPX* exposes special API functions that allow one to create, manage, and read the counter data, and release instances of performance counters. Performance Counter instances are accessed by name, and these names have a predefined structure which is described in the section *Performance counter names*. The advantage of this is that any Performance Counter can be accessed remotely (from a different *locality*) or locally (from the same *locality*). Moreover, since all counters expose their data using the same API, any code consuming counter data can be utilized to access arbitrary system information with minimal effort.

Counter data may be accessed in real time. More information about how to consume counter data can be found in the section *Consuming performance counter data*.

All *HPX* applications provide command line options related to performance counters, such as the ability to list available counter types, or periodically query specific counters to be printed to the screen or save them in a file. For more information, please refer to the section *HPX Command Line Options*.

Performance counter names

All Performance Counter instances have a name uniquely identifying each instance. This name can be used to access the counter, retrieve all related meta data, and to query the counter data (as described in the section *Consuming performance counter data*). Counter names are strings with a predefined structure. The general form of a countername is:

```
/objectname{full_instancename}/countername@parameters
```

where `full_instancename` could be either another (full) counter name or a string formatted as:

```
parentinstancename#parentindex/instancename#instanceindex
```

Each separate part of a countername (e.g., `objectname`, `countername` `parentinstancename`, `instancename`, and `parameters`) should start with a letter (`'a'...'z'`, `'A'...'Z'`) or an underscore character (`'_'`), optionally followed by letters, digits (`'0'...'9'`), hyphen (`'-'`), or underscore characters. Whitespace is not allowed inside a counter name. The characters `'/'`, `'{'`, `'}'`, `'#'` and `'@'` have a special meaning and are used to delimit the different parts of the counter name.

The parts `parentinstanceindex` and `instanceindex` are integers. If an index is not specified, *HPX* will assume a default of `-1`.

Two counter name examples

This section gives examples of both simple counter names and aggregate counter names. For more information on simple and aggregate counter names, please see *Performance counter instances*.

An example of a well-formed (and meaningful) simple counter name would be:

```
/threads{locality#0/total}/count/cumulative
```

This counter returns the current cumulative number of executed (retired) *HPX* threads for the *locality* `0`. The counter type of this counter is `/threads/count/cumulative` and the full instance name is `locality#0/total`. This counter type does not require an `instanceindex` or `parameters` to be specified.

In this case, the `parentindex` (the `'0'`) designates the *locality* for which the counter instance is created. The counter will return the number of *HPX* threads retired on that particular *locality*.

Another example for a well formed (aggregate) counter name is:

```
/statistics{/threads{locality#0/total}/count/cumulative}/average@500
```

This counter takes the simple counter from the first example, samples its values every `500` milliseconds, and returns the average of the value samples whenever it is queried. The counter type of this counter is `/statistics/average` and the instance name is the full name of the counter for which the values have to be averaged. In this case, the `parameters` (the `'500'`) specify the sampling interval for the averaging to take place (in milliseconds).

Performance counter types

Every performance counter belongs to a specific performance counter type which classifies the counters into groups of common semantics. The type of a counter is identified by the `objectname` and the `countername` parts of the name.

```
/objectname/countername
```

When an application starts *HPX* will register all available counter types on each of the localities. These counter types are held in a special performance counter registration database, which can be used to retrieve the meta data related to a counter type and to create counter instances based on a given counter instance name.

Performance counter instances

The `full_instancename` distinguishes different counter instances of the same counter type. The formatting of the `full_instancename` depends on the counter type. There are two types of counters: simple counters, which usually generate the counter values based on direct measurements, and aggregate counters, which take another counter and transform its values before generating their own counter values. An example for a simple counter is given *above*: counting retired *HPX* threads. An aggregate counter is shown as an example *above* as well: calculating the average of the underlying counter values sampled at constant time intervals.

While simple counters use instance names formatted as `parentinstancename#parentindex/instancename#instanceindex`, most aggregate counters have the full counter name of the embedded counter as their instance name.

Not all simple counter types require specifying all four elements of a full counter instance name; some of the parts (`parentinstancename`, `parentindex`, `instancename`, and `instanceindex`) are optional for specific counters. Please refer to the documentation of a particular counter for more information about the formatting requirements for the name of this counter (see *Existing HPX performance counters*).

The `parameters` are used to pass additional information to a counter at creation time. They are optional, and they fully depend on the concrete counter. Even if a specific counter type allows additional parameters to be given, those usually are not required as sensible defaults will be chosen. Please refer to the documentation of a particular counter for more information about what parameters are supported, how to specify them, and what default values are assumed (see also *Existing HPX performance counters*).

Every *locality* of an application exposes its own set of performance counter types and performance counter instances. The set of exposed counters is determined dynamically at application start based on the execution environment of the application. For instance, this set is influenced by the current hardware environment for the *locality* (such as whether the *locality* has access to accelerators), and the software environment of the application (such as the number of OS threads used to execute *HPX* threads).

Using wildcards in performance counter names

It is possible to use wildcard characters when specifying performance counter names. Performance counter names can contain two types of wildcard characters:

- Wildcard characters in the performance counter type
- Wildcard characters in the performance counter instance name

A wildcard character has a meaning which is very close to usual file name wildcard matching rules implemented by common shells (like bash).

Table 2.28: Wildcard characters in the performance counter type

Wildcard	Description
`*`	This wildcard character matches any number (zero or more) of arbitrary characters.
`?`	This wildcard character matches any single arbitrary character.
`[...]`	This wildcard character matches any single character from the list of specified within the square brackets.

Table 2.29: Wildcard characters in the performance counter instance name

Wildcard	Description
`*`	This wildcard character matches any *locality* or any thread, depending on whether it is used for `locality#*` or `worker-thread#*`. No other wildcards are allowed in counter instance names.

Consuming performance counter data

You can consume performance data using either the command line interface, the *HPX* application or the *HPX* API. The command line interface is easier to use, but it is less flexible and does not allow one to adjust the behaviour of your application at runtime. The command line interface provides a convenience abstraction but simplified abstraction for querying and logging performance counter data for a set of performance counters.

Consuming performance counter data from the command line

HPX provides a set of predefined command line options for every application that uses `hpx::init` for its initialization. While there are many more command line options available (see *HPX Command Line Options*), the set of options related to performance counters allows one to list existing counters, and query existing counters once at application termination or repeatedly after a constant time interval.

The following table summarizes the available command line options:

Table 2.30: *HPX* Command Line Options Related to Performance Counters

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

While the options `--hpx:list-counters` and `--hpx:list-counter-infos` give a short list of all available counters, the full documentation for those can be found in the section *Existing HPX performance counters*.

A simple example

All of the commandline options mentioned above can be tested using the `hello_world_distributed` example.

Listing all available counters `hello_world_distributed --hpx:list-counters` yields:

```
List of available counter instances (replace * below with the appropriate
sequence number)
-------------------------------------------------------------------------
/agas/count/allocate /agas/count/bind /agas/count/bind_gid
/agas/count/bind_name ... /threads{locality#*/allocator#*}/count/objects
/threads{locality#*/total}/count/stack-recycles
/threads{locality#*/total}/idle-rate
/threads{locality#*/worker-thread#*}/idle-rate
```

Providing more information about all available counters, `hello_world_distributed --hpx:list-counter-infos` yields:

```
Information about available counter instances (replace * below with the
appropriate sequence number)
-----------------------------------------------------------------------------
fullname: /agas/count/allocate helptext: returns the number of invocations of
the AGAS service 'allocate' type: counter_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:

```
hello_world_distributed \
    --hpx:print-counter=/threads{locality#0/total}/count/cumulative \
    --hpx:print-counter=/agas{locality#0/total}/count/bind \
    --hpx:print-counter-interval=20
```

yields for instance (leaving off the actual console output of the `hello_world_distributed` example for brevity):

```
threads{locality#0/total}/count/cumulative,1,0.002409,[s],22
agas{locality#0/total}/count/bind,1,0.002542,[s],9
threads{locality#0/total}/count/cumulative,2,0.023002,[s],41
agas{locality#0/total}/count/bind,2,0.023557,[s],10
threads{locality#0/total}/count/cumulative,3,0.037514,[s],46
agas{locality#0/total}/count/bind,3,0.038679,[s],10
```

The command `--hpx:print-counter-destination=<file>` will redirect all counter data gathered to the specified file name, which avoids cluttering the console output of your application.

The command line option `--hpx:print-counter` supports using a limited set of wildcards for a (very limited) set of use cases. In particular, all occurrences of `#*` as in `locality#*` and in `worker-thread#*` will be automatically expanded to the proper set of performance counter names representing the actual environment for the executed program. For instance, if your program is utilizing four worker threads for the execution of *HPX* threads (see command line option *`--hpx:threads`*) the following command line

```
hello_world_distributed \
    --hpx:threads=4 \
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative
```

will print the value of the performance counters monitoring each of the worker threads:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 0
hello world from OS-thread 2 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.0025214,[s],27
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.0025453,[s],33
/threads{locality#0/worker-thread#2}/count/cumulative,1,0.0025683,[s],29
/threads{locality#0/worker-thread#3}/count/cumulative,1,0.0025904,[s],33
```

The command `--hpx:print-counter-format` takes values `csv` and `csv-short` to generate CSV formatted counter values with a header.

With format as csv:

```
hello_world_distributed \
    --hpx:threads=2 \
    --hpx:print-counter-format csv \
    --hpx:print-counter /threads{locality#*/total}/count/cumulative \
    --hpx:print-counter /threads{locality#*/total}/count/cumulative-phases
```

will print the values of performance counters in CSV format with the full countername as a header:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
/threads{locality#*/total}/count/cumulative,/threads{locality#*/total}/count/
↪cumulative-phases
39,93
```

With format csv-short:

```
hello_world_distributed \
    --hpx:threads 2 \
    --hpx:print-counter-format csv-short \
    --hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \
    --hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases
```

will print the values of performance counters in CSV format with the short countername as a header:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
cumulative,phases
39,93
```

With format csv and csv-short when used with `--hpx:print-counter-interval`:

```
hello_world_distributed \
    --hpx:threads 2 \
    --hpx:print-counter-format csv-short \
    --hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \
    --hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases \
    --hpx:print-counter-interval 5
```

will print the header only once repeating the performance counter value(s) repeatedly:

```
cum,phases
25,42
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
44,95
```

The command `--hpx:no-csv-header` can be used with `--hpx:print-counter-format` to print performance counter values in CSV format without any header:

```
hello_world_distributed \
--hpx:threads 2 \
--hpx:print-counter-format csv-short \
--hpx:print-counter cumulative,/threads{locality#*/total}/count/cumulative \
--hpx:print-counter phases,/threads{locality#*/total}/count/cumulative-phases \
--hpx:no-csv-header
```

will print:

```
hello world from OS-thread 1 on locality 0
hello world from OS-thread 0 on locality 0
37,91
```

Consuming performance counter data using the *HPX* API

HPX provides an API that allows users to discover performance counters and to retrieve the current value of any existing performance counter from any application.

Discover existing performance counters

Retrieve the current value of any performance counter

Performance counters are specialized *HPX* components. In order to retrieve a counter value, the performance counter needs to be instantiated. *HPX* exposes a client component object for this purpose:

```
hpx::performance_counters::performance_counter counter(std::string const& name);
```

Instantiating an instance of this type will create the performance counter identified by the given `name`. Only the first invocation for any given counter name will create a new instance of that counter. All following invocations for a given counter name will reference the initially created instance. This ensures that at any point in time there is never more than one active instance of any of the existing performance counters.

In order to access the counter value (or to invoke any of the other functionality related to a performance counter, like `start`, `stop` or `reset`) member functions of the created client component instance should be called:

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

```
// 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:

```
std::int64_t some_performance_data(bool reset);
```

The argument `bool reset` (which is supplied by the runtime system when the function is invoked) specifies whether the counter value should be reset after evaluating the current value (if applicable).

For instance, here is such a function returning how often it was invoked:

```
// The atomic variable 'counter' ensures the thread safety of the counter.
boost::atomic<std::int64_t> counter(0);

std::int64_t some_performance_data(bool reset)
{
    std::int64_t result = ++counter;
    if (reset)
        counter = 0;
    return result;
}
```

This example function exposes a linearly-increasing value as our performance data. The value is incremented on each invocation, i.e., each time a consumer requests the counter data of this performance counter.

The next step in exposing this counter to the runtime system is to register the function as a new raw counter type using the *HPX* API function *hpx::performance_counters::install_counter_type*. A counter type represents certain common characteristics of counters, like their counter type name and any associated description information. The following snippet shows an example of how to register the function `some_performance_data`, which is shown above, for a counter type named `"/test/data"`. This registration has to be executed before any consumer instantiates, and queries an instance of this counter type:

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

void register_counter_type()
{
    // Call the HPX API function to register the counter type.
    hpx::performance_counters::install_counter_type(
        "/test/data",                                   // counter type name
        &some_performance_data,                         // function providing counter␣
↪data
        "returns a linearly increasing counter value"   // description text (optional)
        ""                                              // unit of measure (optional)
    );
}
```

Now it is possible to instantiate a new counter instance based on the naming scheme `"/test{locality#*/total}/data"` where `*` is a zero-based integer index identifying the *locality* for which the counter instance should be accessed. The function *hpx::performance_counters::install_counter_type* enables users to instantiate exactly one counter instance for each *locality*. Repeated requests to instantiate such a counter will return the same instance, i.e., the instance created for the first request.

If this counter needs to be accessed using the standard *HPX* command line options, the registration has to be performed during application startup, before `hpx_main` is executed. The best way to achieve this is to register an *HPX* startup function using the API function *hpx::register_startup_function* before calling `hpx::init` to initialize the runtime system:

```
int main(int argc, char* argv[])
{
    // By registering the counter type we make it available to any consumer
    // who creates and queries an instance of the type "/test/data".
    //
    // This registration should be performed during startup. The
    // function 'register_counter_type' should be executed as an HPX thread right
    // before hpx_main is executed.
    hpx::register_startup_function(&register_counter_type);

    // Initialize and run HPX.
    return hpx::init(argc, argv);
}
```

Please see the code in `simplest_performance_counter.cpp` for a full example demonstrating this functionality.

Implementing a full performance counter

Sometimes, the simple way of exposing a single value as a performance counter is not sufficient. For that reason, *HPX* provides a means of implementing full performance counters which support:

- Retrieving the descriptive information about the performance counter
- Retrieving the current counter value
- Resetting the performance counter (value)
- Starting the performance counter
- Stopping the performance counter
- Setting the (initial) value of the performance counter

Every full performance counter will implement a predefined interface:

```
//  Copyright (c) 2007-2020 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/functional/bind_front.hpp>
#include <hpx/futures/future.hpp>
#include <hpx/modules/execution.hpp>
#include <hpx/runtime/components/client_base.hpp>

#include <hpx/performance_counters/counters_fwd.hpp>
#include <hpx/performance_counters/server/base_performance_counter.hpp>

#include <string>
#include <utility>
#include <vector>
```

(continues on next page)

(continued from previous page)

```
///////////////////////////////////////////////////////////////////////////////
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(
            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);

        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();
```

(continues on next page)

(continued from previous page)

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

    // Return all counters matching the given name (with optional wild cards).
    HPX_EXPORT std::vector<performance_counter> discover_counters(
        std::string const& name, error_code& ec = throws);
}}    // namespace hpx::performance_counters
```

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_link hpx/performance_counters/base_performance_counter.hpp..hpx/performance_count as:

```
//  Copyright (c) 2007-2018 Hartmut Kaiser
//
//  SPDX-License-Identifier: BSL-1.0
//  Distributed under the Boost Software License, Version 1.0. (See accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/config.hpp>
#include <hpx/actions_base/component_action.hpp>
#include <hpx/performance_counters/counters.hpp>
#include <hpx/performance_counters/server/base_performance_counter.hpp>
#include <hpx/runtime/components/component_type.hpp>
#include <hpx/runtime/components/server/component_base.hpp>

///////////////////////////////////////////////////////////////////////////////
//[performance_counter_base_class
namespace hpx { namespace performance_counters {
    template <typename Derived>
    class base_performance_counter;
}}    // namespace hpx::performance_counters
//]

///////////////////////////////////////////////////////////////////////////////
namespace hpx { namespace performance_counters {
    template <typename Derived>
    class base_performance_counter
      : public hpx::performance_counters::server::base_performance_counter
      , public hpx::components::component_base<Derived>
    {
    private:
        typedef hpx::components::component_base<Derived> base_type;

    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();
```

(continues on next page)

(continued from previous page)

```
        }
    };
}}    // namespace hpx::performance_counters
```

The single template parameter is expected to receive the type of the derived class implementing the performance counter. In the Sine example this looks like:

```
//  Copyright (c) 2007-2012 Hartmut Kaiser
//
//  SPDX-License-Identifier: BSL-1.0
//  Distributed under the Boost Software License, Version 1.0. (See accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#pragma once

#include <hpx/hpx.hpp>
#include <hpx/include/lcos_local.hpp>
#include <hpx/include/performance_counters.hpp>
#include <hpx/include/util.hpp>

#include <cstdint>

namespace performance_counters { namespace sine { namespace server
{
    ///////////////////////////////////////////////////////////////////////////
    //[sine_counter_definition
    class sine_counter
      : public hpx::performance_counters::base_performance_counter<sine_counter>
    //]
    {
    public:
        sine_counter() : current_value_(0), evaluated_at_(0) {}
        explicit sine_counter(
            hpx::performance_counters::counter_info const& info);

        /// This function will be called in order to query the current value of
        /// this performance counter
        hpx::performance_counters::counter_value get_counter_value(bool reset);

        /// The functions below will be called to start and stop collecting
        /// counter values from this counter.
        bool start();
        bool stop();

        /// finalize() will be called just before the instance gets destructed
        void finalize();

    protected:
        bool evaluate();

    private:
        typedef hpx::lcos::local::spinlock mutex_type;

        mutable mutex_type mtx_;
        double current_value_;
        std::uint64_t evaluated_at_;
```

(continues on next page)

(continued from previous page)

```
        hpx::util::interval_timer timer_;
    };
}}}
```

i.e., the type `sine_counter` is derived from the base class passing the type as a template argument (please see `simplest_performance_counter.cpp` for the full source code of the counter definition). For more information about this technique (called Curiously Recurring Template Pattern - CRTP), please see for instance the corresponding Wikipedia article[196]. This base class itself is derived from the `performance_counter` interface described above.

Additionally, a full performance counter implementation not only exposes the actual value but also provides information about:

- The point in time a particular value was retrieved.
- A (sequential) invocation count.
- The actual counter value.
- An optional scaling coefficient.
- Information about the counter status.

Existing *HPX* performance counters

The *HPX* runtime system exposes a wide variety of predefined performance counters. These counters expose critical information about different modules of the runtime system. They can help determine system bottlenecks and fine-tune system and application performance.

[196] http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Table 2.31: AGAS performance counters

Counter type	Counter instance formatting	Description	Parameters
/agas/count/<agas_service> where: <agas_service> is one of the following: *primary namespace services*: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate, begin_migration, end_migration *component namespace services*: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type *locality namespace services*: free, localities, num_localities, num_threads, resolve_locality, resolved_localities *symbol namespace services*: bind, resolve, unbind, iterate_names, on_symbol_namespace_event	<agas_instance>/total where: <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the locality hosting the *AGAS* service). The value for * can be any *locality* id for the following <agas_service>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, bin, resolve, unbind, and iterate_names (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).	None	Returns the total number of invocations of the specified *AGAS* service since its creation.
/agas/<agas_service_category>/count where: <agas_service_category> is one of the following: primary, locality, component or symbol	<agas_instance>/total where: <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the *locality* hosting the *AGAS* service). Except for <agas_service_category>, primary or symbol for which the value for * can be any *locality* id (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).	None	Returns the overall total number of invocations of all *AGAS* services provided by the given *AGAS* service category since its creation.
agas/time/<agas_service> where: <agas_service> is one of the following: *primary namespace services*: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, allocate begin_migration, end_migration *component namespace services*: bind_prefix, bind_name, resolve_id, unbind_name, iterate_types, get_component_typename, num_localities_type *locality namespace services*: free, localities, num_localities, num_threads, resolve_locality, resolved_localities *symbol namespace services*: bind, resolve, unbind, iterate_names,	<agas_instance>/total where: <agas_instance> is the name of the *AGAS* service to query. Currently, this value will be locality#0 where 0 is the root *locality* (the id of the *locality* hosting the *AGAS* service). The value for * can be any *locality* id for the following <agas_service>: route, bind_gid, resolve_gid, unbind_gid, increment_credit, decrement_credit, bin, resolve, unbind, and iterate_names (only the primary and symbol *AGAS* service components live on all localities, whereas all other *AGAS* services are available on locality#0 only).	None	Returns the overall execution time of the specified *AGAS* service since its creation (in nanoseconds).

Table 2.32: Parcel layer performance counters

Counter type	Counter instance formatting	Description	Parameters
/data/count/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi	locality#*/total where: * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.	Returns the overall number of raw (uncompressed) bytes sent or received (see <operation, e.g. en or eceived) for the specified <connection_type>. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default, the corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake variables used to configure HPX* for more details.	None
/data/time/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi	locality#*/total where: * is the *locality* id of the *locality* the total transmission time should be queried for. The *locality* id is a (zero based) number identifying the *locality*.	Returns the total time (in nanoseconds) between the start of each asynchronous transmission operation and the end of the corresponding operation for the specified <connection_type> the given *locality* (see <operation, e.g. en or eceived). The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default, the corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake variables used to configure HPX* for more details.	None
/serialize/count/<connection_type>/<operation> where: <operation> is one of the following: sent, received <connection_type> is one of the following: tcp, mpi	locality#*/total where: * is the *locality* id of the *locality* the overall number of transmitted bytes should be queried for. The *locality* id is a (zero based) number identifying the *locality*.	Returns the overall number of bytes transferred (see <operation>, e.g. sent or received possibly compressed) for the specified <connection_type> by the given *locality*. The performance counters for the connection type mpi are available only if the compile time constant HPX_HAVE_PARCELPORT_MPI was defined while compiling the *HPX* core library (which is not defined by default, the corresponding cmake configuration constant is HPX_WITH_PARCELPORT_MPI. Please see *CMake variables used to configure HPX* for more details.	If the configure-time option -DHPX_WITH_PARCELPORT_ACTIO 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.
/serialize/time/<connection_type>/<operation> where:	locality#*/total where: * is the *locality* id of the	Returns the overall time spent performing outgoing data serialization for the specified <connection_type> on the given *locality* (see <operation, e.g. sent or received). The performance counters for the connection	If the configure-time option -DHPX_WITH_PARCELPORT_ACTIO was specified, this counter allows one

Table 2.33: Thread manager performance counters

Counter type	Counter instance formatting	Description	Parameters
`/threads/count/cumulative`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the overall number of retired *HPX*-threads should be queried for. The *locality* id (given by `*` 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.	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`).	None

continues on next page

[197] A message can potentially consist of more than one *parcel*.

Table 2.33 – continued from previous page

/threads/time/average	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average time spent executing one *HPX*-thread should be queried for. The *locality* id (given by * is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the average time spent executing one *HPX*-thread should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.	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].	None

continues on next page

Table 2.33 – continued from previous page

/threads/time/average-overhead	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average overhead spent executing one *HPX*-thread should be queried for. The *locality* id (given by * 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

continues on next page

Table 2.33 – 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 2.33 – continued from previous page

`/threads/time/average-phase`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the average time spent executing one *HPX*-thread phase (invocation) should be queried for. The *locality* id (given by `*` 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.	Returns the average time spent executing one *HPX*-thread phase (invocation) on the given *locality* since application start. If the instance name is `total` the counter returns the average time spent executing one *HPX*-thread phase (invocation) for all worker threads (cores) on that *locality*. If the instance name is `worker-thread#*` the counter will return the average time spent executing one *HPX*-thread phase for all worker threads separately. This counter is available only if the configuration time constants `HPX_WITH_THREAD_CUMULATIVE_COUNTS` (default: `ON`) and `HPX_WITH_THREAD_IDLE_RATES` are set to `ON` (default: `OFF`). The unit of measure for this counter is nanosecond [ns].	None

continues on next page

Table 2.33 – continued from previous page

/threads/time/ average-phase-overhead	locality#*/total or locality#*/ worker-thread#* or locality#*/ pool#*/ worker-thread#* where: locality#* is defining the *locality* for which the average time overhead executing one *HPX*-thread phase (invocation) should be queried for. The *locality* id (given by * 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].	None

continues on next page

Table 2.33 – continued from previous page

`/threads/time/overall`	`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.	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].	None

continues on next page

Table 2.33 – 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 2.33 – continued from previous page

`/threads/time/cumulative-overheads`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the overall overhead time incurred by executing all *HPX*-threads should be queried for. The *locality* id (given by `*` is a (zero based) number identifying the *locality*. `pool#*` is defining the pool for which the current value of the idle-loop counter should be queried for. `worker-thread#*` is defining the worker thread for which the the overall overhead time incurred by executing all *HPX*-threads should be queried for. The worker thread number (given by the `*` is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *`--hpx:threads`*. If no pool-name is specified the counter refers to the 'default' pool.	Returns the overall overhead time incurred executing all *HPX*-threads on the given *locality* since application start. If the instance name is `total` the counter returns the overall overhead time incurred executing all *HPX*-threads for all worker threads (cores) on that *locality*. If the instance name is `worker-thread#*` the counter will return the overall overhead time incurred executing all *HPX*-threads for all worker threads separately. This counter is available only if the configuration time constants `HPX_THREAD_MAINTAIN_CUMULATIVE_COUNTS` (default: `ON`) and `HPX_THREAD_MAINTAIN_IDLE_RATES` are set to `ON` (default: `OFF`). The unit of measure for this counter is nanosecond [ns].	None

continues on next page

Table 2.33 – continued from previous page

`threads/count/instantaneous/<thread-state>` where: `<thread-state>` is one of the following: `all`, `active`, `pending`, `suspended`, `terminated`, `staged`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the current number of threads with the given state should be queried for. The *locality* id (given by `*` 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.	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

continues on next page

Table 2.33 – continued from previous page

`threads/wait-time/<thread-state>` where: `<thread-state>` is one of the following: `pending` `staged`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the average wait time of *HPX*-threads (pending) or thread descriptions (staged) with the given state should be queried for. The *locality* id (given by `*` is a (zero based) number identifying the *locality*. `pool#*` is defining the pool for which the current value of the idle-loop counter should be queried for. `worker-thread#*` is defining the worker thread for which the average wait time for the given state should be queried for. The worker thread number (given by the `*` is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *`--hpx:threads`*. If no pool-name is specified the counter refers to the 'default' pool. The `staged` thread state refers to the wait time of registered tasks before they are converted into thread objects, while the `pending` thread state refers to the wait time of threads in any of the scheduling queues.	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

continues on next page

Table 2.33 – continued from previous page

`/threads/idle-rate`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the average idle rate of all (or one) worker threads should be queried for. The *locality* id (given by `*` 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.	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`).	None

continues on next page

Table 2.33 – continued from previous page

`/threads/creation-idle-rate`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the average creation idle rate of all (or one) worker threads should be queried for. The *locality* id (given by `*` 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.	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

continues on next page

Table 2.33 – continued from previous page

/threads/cleanup-idle-rate	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the average cleanup idle rate of all (or one) worker threads should be queried for. The *locality* id (given by * is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the averaged cleanup idle rate should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.	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.	None

continues on next page

Table 2.33 – continued from previous page

/threadqueue/length	locality#*/total or locality#*/worker-thread#* or locality#*/pool#*/worker-thread#* where: locality#* is defining the *locality* for which the current length of all thread queues in the scheduler for all (or one) worker threads should be queried for. The *locality* id (given by * is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the current length of all thread queues in the scheduler should be queried for. The worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.	Returns the overall length of all queues for the given worker thread(s) on the given *locality*.	None
/threads/count/stack-unbinds	locality#*/total where: * is the *locality* id of the *locality* the unbind (madvise) operations should be queried for. The *locality* id is a (zero based) number identifying the *locality*.	Returns the total number of *HPX*-thread unbind (madvise) operations performed for the referenced *locality*. Note that this counter is not available on Windows based platforms.	None

continues on next page

Table 2.33 – continued from previous page

/threads/count/stack-recycles	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*.	Returns the total number of *HPX*-thread recycling operations performed.	None
/threads/count/stolen-from-pending	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*.	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).	None

continues on next page

Table 2.33 – 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 2.33 – continued from previous page

`/threads/count/pending-accesses`	`locality#*/total` or `locality#*/worker-thread#*` or `locality#*/pool#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the number of pending queue accesses of all (or one) worker threads should be queried for. The *locality* id (given by `*` is a (zero based) number identifying the *locality* `pool#*` is defining the pool for which the current value of the idle-loop counter should be queried for. `worker-thread#*` is defining the worker thread for which the number of pending queue accesses should be queried for. The worker thread number (given by the `*` is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *`--hpx:threads`*. If no pool-name is specified the counter refers to the 'default' pool.	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 2.33 – 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 2.33 – 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 2.33 – continued from previous page

/threads/count/stolen-to-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 to the staged queue of all (or one) worker threads should be queried for. The *locality* id (given by * is a (zero based) number identifying the *locality*. pool#* is defining the pool for which the current value of the idle-loop counter should be queried for. worker-thread#* is defining the worker thread for which the number of *HPX*-threads stolen to the staged queue should be queried for. The worker thread number (given by the * is a (zero based) worker thread number (given by the * is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*. If no pool-name is specified the counter refers to the 'default' pool.	Returns the total number of *HPX*-threads 'stolen' to the staged thread queue of a neighboring worker thread (these threads are executed by a different worker thread than they were initially scheduled on). This counter is available only if the configuration time constant HPX_WITH_THREAD_STEALING_COUNTS is set to ON (default: ON).	None

continues on next page

Table 2.33 – continued from previous page

`/threads/count/objects`	`locality#*/total` or `locality#*/allocator#*` where: `locality#*` is defining the *locality* for which the current (cumulative) number of all created *HPX*-thread objects should be queried for. The *locality* id (given by `*` is a (zero based) number identifying the *locality*. `allocator#*` is defining the number of the allocator instance using which the threads have been created. *HPX* uses a varying number of allocators to create (and recycle) *HPX*-thread objects, most likely these counters are of use for debugging purposes only. The allocator id (given by `*` is a (zero based) number identifying the allocator to query.	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.	None
`/scheduler/utilization/instantaneous`	`locality#*/total` where: `locality#*` is defining the *locality* for which the current (instantaneous) scheduler utilization queried for. The *locality* id (given by `*` is a (zero based) number identifying the *locality*.	**Returns the total (instantaneous) scheduler utilization. This is the** current percentage of scheduler threads executing *HPX* threads.	Percent

continues on next page

Table 2.33 – 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 2.33 – continued from previous page

`/threads/busy-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 busy-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 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.	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 2.33 – continued from previous page

/threads/time/background-work-duration	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.	Returns the overall time spent performing background work on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns].	None

continues on next page

Table 2.33 – 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 2.33 – continued from previous page

/threads/time/background-send-duration	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work related to sending parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work related to sending parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.	Returns the overall time spent performing background work related to sending parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcelport only.	None

continues on next page

Table 2.33 – continued from previous page

/threads/background-send-overhead	locality#*/total locality#*/worker-thread#* where: locality#* is defining the locality for which the background overhead related to sending parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the background overhead related to sending parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.	Returns the background overhead related to sending parcels on the given locality since application start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcelport only.	None

continues on next page

Table 2.33 – continued from previous page

/threads/time/background-receive-duration	locality#*/total or locality#*/worker-thread#* where: locality#* is defining the locality for which the overall time spent performing background work related to receiving parcels should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the overall time spent performing background work related to receiving parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.	Returns the overall time spent performing background work related to receiving parcels on the given locality since application start. If the instance name is total the counter returns the overall time spent performing background work for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return the overall time spent performing background work for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure for this counter is nanosecond [ns]. This counter will currently return meaningful values for the MPI parcelport only.	None

continues on next page

Table 2.33 – continued from previous page

/threads/ background-receive-overhead	locality#*/total or locality#*/ worker-thread#* where: locality#* is defining the locality for which the background overhead related to receiving should be queried for. The locality id (given by *) is a (zero based) number identifying the locality. worker-thread#* is defining the worker thread for which the background overhead related to receiving parcels should be queried for. The worker thread number (given by the *) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *--hpx:threads*.	Returns the background overhead related to receiving parcels on the given locality since application start. If the instance name is total the counter returns the background overhead for all worker threads (cores) on that locality. If the instance name is worker-thread#* the counter will return background overhead for all worker threads separately. This counter is available only if the configuration time constants HPX_WITH_BACKGROUND_THREAD_COUNTERS (default: OFF) and HPX_WITH_THREAD_IDLE_RATES are set to ON (default: OFF). The unit of measure displayed for this counter is 0.1%. This counter will currently return meaningful values for the MPI parcelport only.	None

Table 2.34: General performance counters exposing characteristics of localities

Counter type	Counter instance formatting	Description	Parameters
`/runtime/count/component`	`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*.	Returns the overall number of currently active components of the specified type on the given *locality*.	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*.
`/runtime/count/action-invocation`	`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*.	Returns the overall (local) invocation count of the specified action type on the given *locality*.	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*.
`/runtime/count/remote-action-invocation`	`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*.	Returns the overall (remote) invocation count of the specified action type on the given *locality*.	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*.
`/runtime/uptime`	`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*.	Returns the overall time since application start on the given *locality* in nanoseconds.	None
`/runtime/memory/virtual`	`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*.	Returns the amount of virtual memory currently allocated by the referenced *locality* (in bytes).	None
`/runtime/memory/resident`	`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*.	Returns the amount of resident memory currently allocated by the referenced *locality* (in bytes).	None
`/runtime/memory/total`	`locality#*/total` where:	**Returns the total available memory for use by the referenced** *locality* (in bytes).	None

Table 2.35: Performance counters exposing PAPI hardware counters

Counter type	Counter instance formatting	Description	Parameters
`/papi/<papi_event>` where: `<papi_event>` is the name of the PAPI event to expose as a performance counter (such as `PAPI_SR_INS`). Note that the list of available PAPI events changes depending on the used architecture. For a full list of available PAPI events and their (short) description use the `--hpx:list-counters` and `--hpx:papi-event-info=all` command line options.	`locality#*/total` or `locality#*/worker-thread#*` where: `locality#*` is defining the *locality* for which the current current accumulated value of all busy-loop counters of all worker threads should be queried. The *locality* id (given by `*`) is a (zero based) number identifying the *locality*. `worker-thread#*` is defining the worker thread for which the current value of the busy-loop counter should be queried for. The worker thread number (given by the `*`) is a (zero based) worker thread number (given by the `*`) is a (zero based) number identifying the worker thread. The number of available worker threads is usually specified on the command line for the application using the option *`--hpx:threads`*.	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`).	None

Table 2.36: Performance counters for general statistics

Counter type	Counter instance formatting	Description	Parameters
/statistics/average	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name).	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`.
/statistics/rolling_average	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current rolling average (mean) value calculated based on the values queried from the underlying counter (the one specified as the instance name).	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`.
/statistics/stddev	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current standard deviation (stddev) value calculated based on the values queried from the underlying counter (the one specified as the instance name).	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`.
/statistics/rolling_stddev	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current rolling variance (stddev) value calculated based on the values queried from the underlying counter (the one specified as the instance name).	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`.
/statistics/median	Any full performance counter name. The referenced performance counter is queried at fixed time intervals as specified by the first parameter.	Returns the current (statistically estimated) median value calculated based on the values queried from the underlying counter (the one specified as the instance name).	Any parameter will be interpreted as a list of up to two comma separated (integer) values, where the first is the time interval (in milliseconds) at which the underlying counter should be queried. If no value is specified, the counter will assume `1000` [ms] as the default. The second value can be either `0` or `1` and specifies whether the underlying counter should be reset during evaluation `1` or not `0`. The default value is `0`.

Table 2.37: Performance counters for elementary arithmetic operations

Counter type	Counter instance formatting	Description	Parameters
`/arithmetics/add`	None	Returns the sum calculated based on the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/subtract`	None	Returns the difference calculated based on the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/multiply`	None	Returns the product calculated based on the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/divide`	None	Returns the result of division of the values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/mean`	None	Returns the average value of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/variance`	None	Returns the standard deviation of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/median`	None	Returns the median value of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/min`	None	Returns the minimum value of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/max`	None	Returns the maximum value of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.
`/arithmetics/count`	None	Returns the count value of all values queried from the underlying counters (the ones specified as the parameters).	The parameter will be interpreted as a comma separated list of full performance counter names which are queried whenever this counter is accessed. Any wildcards in the counter names will be expanded.

Note: The `/arithmetics` counters can consume an arbitrary number of other counters. For this reason those have to be specified as parameters (a comma separated list of counters appended after a `'@'`). For instance:

```
./bin/hello_world_distributed -t2 \
```

(continues on next page)

(continued from previous page)

```
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
    --hpx:print-counter=/arithmetics/add@/threads{locality#0/worker-thread#*}/count/
↪cumulative
hello world from OS-thread 0 on locality 0
hello world from OS-thread 1 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.515640,[s],25
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.515520,[s],36
/arithmetics/add@/threads{locality#0/worker-thread#*}/count/cumulative,1,0.516445,[s],
↪64
```

Since all wildcards in the parameters are expanded, this example is fully equivalent to specifying both counters separately to `/arithmetics/add`:

```
./bin/hello_world_distributed -t2 \
    --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative \
    --hpx:print-counter=/arithmetics/add@\
        /threads{locality#0/worker-thread#0}/count/cumulative,\
        /threads{locality#0/worker-thread#1}/count/cumulative
```

Table 2.38: Performance counters tracking parcel coalescing

Counter type	Counter instance formatting	Description	Parameters
/coalescing/count/parcels	locality#*/total where: * is the *locality* id of the *locality* the number of parcels for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.	Returns the number of parcels handled by the message handler associated with the action which is given by the counter parameter.	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*.
/coalescing/count/messages	locality#*/total where: * is the *locality* id of the *locality* the number of messages for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.	Returns the number of messages generated by the message handler associated with the action which is given by the counter parameter.	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*.
/coalescing/count/average-parcels-per-message	locality#*/total where: * is the *locality* id of the *locality* the number of messages for the given action should be queried for. The *locality* id is a (zero based) number identifying the *locality*.	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.	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*
/coalescing/time/average-parcel-arrival	locality#*/total where: * is the *lo-*	Returns the average time between arriving parcels for the action which is given by the counter parameter.	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

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[198], which is a framework for application profiling using task timers and various performance counters. It can be added as a `git` submodule by turning on the option *`HPX_WITH_APEX:BOOL`* during CMake configuration. TAU[199] 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[200] for detailed instructions on using APEX with *HPX*.

2.5.11 *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.

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`.

198 https://khuck.github.io/xpress-apex/
199 https://www.cs.uoregon.edu/research/tau/home.php
200 https://khuck.github.io/xpress-apex/usage/#hpx-louisiana-state-university

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`.

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[201] 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[202] communication. MPI[203] tasks will then always run on the same thread, instead of potentially being stuck in a queue behind other threads.

Note that *HPX* thread pools are completely independent from each other in the sense that task stealing will never happen between different thread pools. However, tasks running on a particular thread pool can schedule tasks on another thread pool.

Note: It is simpler in some situations to schedule important tasks with high priority instead of using a separate thread pool.

Using the resource partitioner

The *`hpx::resource::partitioner`* is now created during *HPX* runtime initialization without explicit action needed from the user. To specify some of the initialization parameters you can use the *`hpx::init_params`*.

```
#include <hpx/hpx_init.hpp>
#include <hpx/resource_partitioner/partitioner.hpp>

int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

int main(int argc, char** argv)
{
    // Setup the init parameters
    hpx::init_params init_args;
    hpx::init(argc, argv, init_args);
}
```

The resource partitioner callback is the interface to add thread pools to the *HPX* runtime and to assign resources to the thread pools. In order to create custom thread pools you can specify the resource partitioner callback *`hpx::init_params::rp_callback`* which will be called once the resource partitioner will be created , see the example below. You can also specify other parameters, see *`hpx::init_params`*.

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

Then, to add resources to the thread pool you can use the *`hpx::resource::partitioner::add_resource`* method. The resource partitioner exposes the hardware topology retrieved using Portable Hardware Locality

201 https://en.wikipedia.org/wiki/Message_Passing_Interface
202 https://en.wikipedia.org/wiki/Message_Passing_Interface
203 https://en.wikipedia.org/wiki/Message_Passing_Interface

(HWLOC)[204] 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:

```
#include <hpx/hpx_init.hpp>
#include <hpx/resource_partitioner/partitioner.hpp>

#include <iostream>

int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

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

    bool one_numa_domain = rp.numa_domains().size() == 1;
    bool skipped_first_pu = false;

    hpx::resource::numa_domain const& d = rp.numa_domains()[0];

    for (const hpx::resource::core& c : d.cores())
    {
        for (const hpx::resource::pu& p : c.pus())
        {
            if (one_numa_domain && !skipped_first_pu)
            {
                skipped_first_pu = true;
                continue;
            }

            rp.add_resource(p, "my-thread-pool");
        }
    }
}

int main(int argc, char* argv[])
{
    // Set the callback to init the thread_pools
    hpx::init_params init_args;
    init_args.rp_callback = &init_resource_partitioner_handler;

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

Note: Whatever processing units not assigned to a thread pool by the time *hpx::init* is called will be added to the default thread pool. It is also possible to explicitly add processing units to the default thread pool, and to create the default thread pool manually (in order to e.g. set the scheduler type).

Tip: The command line option *--hpx:print-bind* is useful for checking that the thread pools have been set up the way you expect.

[204] https://www.open-mpi.org/projects/hwloc/

Difference between the old and new version

In the old version, you had to create an instance of the `resource_partitioner` with `argc` and `argv`.

```
int main(int argc, char** argv)
{
    hpx::resource::partitioner rp(argc, argv);
    hpx::init();
}
```

From *HPX* 1.5.0 onwards, you just pass `argc` and `argv` to `hpx::init()` or `hpx::start()` for the binding options to be parsed by the resource partitioner.

```
int main(int argc, char** argv)
{
    hpx::init_params init_args;
    hpx::init(argc, argv, init_args);
}
```

In the old version, when creating a custom thread pool, you just called the utilities on the resource partitioner instantiated previously.

```
int main(int argc, char** argv)
{
    hpx::resource::partitioner rp(argc, argv);

    rp.create_thread_pool("my-thread-pool");

    bool one_numa_domain = rp.numa_domains().size() == 1;
    bool skipped_first_pu = false;

    hpx::resource::numa_domain const& d = rp.numa_domains()[0];

    for (const hpx::resource::core& c : d.cores())
    {
        for (const hpx::resource::pu& p : c.pus())
        {
            if (one_numa_domain && !skipped_first_pu)
            {
                skipped_first_pu = true;
                continue;
            }

            rp.add_resource(p, "my-thread-pool");
        }
    }

    hpx::init();
}
```

You now specify the resource partitioner callback which will tie the resources to the resource partitioner created during runtime initialization.

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

```

(continues on next page)

(continued from previous 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:

```
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`.

2.5.12 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/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\n";

            // Print all of the available diagnostic information as stored with
            // the exception.
            hpx::cout << "diagnostic information:"
```

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:

```
// 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 << "{stack-trace}: " << hpx::get_error_backtrace(e) << "\n";
```

Working with error codes

Most of the API functions exposed by *HPX* can be invoked in two different modes. By default those will throw an exception on error as described above. However, sometimes it is desirable not to throw an exception in case of an error condition. In this case an object instance of the `hpx::error_code` type can be passed as the last argument to the API function. In case of an error, the error condition will be returned in that `hpx::error_code` instance. The following example demonstrates extracting the full diagnostic information without exception handling:

```
// 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";

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

(continues on next page)

(continued from previous page)

```
            // the exception.
            hpx::cout << "diagnostic information:"
                << hpx::diagnostic_information(ec) << "\n";
```

Note: The error information is transferred back to the invocation site even if it is executed on a different *locality*.

This example show how an error can be handled without having to resolve to exceptions and that the returned *hpx::error_code* instance can be used in a very similar way as the *hpx::exception* type above. Simply pass it to the *hpx::diagnostic_information*, which retrieves all available diagnostic information from the error code instance as a formatted string.

As for handling exceptions, when working with error codes, under certain circumstances it is desirable to output only some of the diagnostics, or to output those using different formatting. For this case, *HPX* exposes a set of lower-level functions usable with error codes as demonstrated in the following code snippet:

```
        // Detailed error reporting using error code
        {
            //[error_handling_diagnostic_elements
            hpx::cout << "Detailed error reporting using error code\n";

            // Create a new error_code instance.
            hpx::error_code ec;

            // If an instance of an error_code is passed as the last argument while
            // invoking the action, the function will not throw in case of an error
            // but store the error information in this error_code instance instead.
            raise_exception_action do_it;
            do_it(hpx::find_here(), ec);

            if (ec) {
                // Print the elements of the diagnostic information separately.
                hpx::cout << "{what}: "        << hpx::get_error_what(ec) << "\n";
                hpx::cout << "{locality-id}: " << hpx::get_error_locality_id(ec) << "\
↪n";
                hpx::cout << "{hostname}: "    << hpx::get_error_host_name(ec) << "\n
↪";
                hpx::cout << "{pid}: "         << hpx::get_error_process_id(ec) << "\n
↪";
                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\n";
                hpx::cout << "{state}: "       << std::hex << hpx::get_error_state(ec)
                    << "\n";
                hpx::cout << "{stack-trace}: " << hpx::get_error_backtrace(ec) << "\n
↪";
                hpx::cout << "{env}: "         << hpx::get_error_env(ec) << "\n";
```

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:

```
// 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.
        hpx::cout << "error code:" << ec.value() << "\n";
```

All functions that retrieve other diagnostic elements from the *hpx::error_code* will fail if called with a lightweight error_code instance.

Utilities in *HPX*

In order to ease the burden of programming, *HPX* provides several utilities to users. The following section documents those facilies.

Checkpoint

See *checkpoint*.

The *HPX* I/O-streams component

The *HPX* I/O-streams subsystem extends the standard C++ output streams `std::cout` and `std::cerr` to work in the distributed setting of an *HPX* application. All of the output streamed to `hpx::cout` will be dispatched to `std::cout` on the console *locality*. Likewise, all output generated from `hpx::cerr` will be dispatched to `std::cerr` on the console *locality*.

Note: All existing standard manipulators can be used in conjunction with `hpx::cout` and `hpx::cerr` 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:

```
//  Copyright (c) 2007-2012 Hartmut Kaiser
//
//  SPDX-License-Identifier: BSL-1.0
//  Distributed under the Boost Software License, Version 1.0. (See accompanying
//  file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

///////////////////////////////////////////////////////////////////////////////
// The purpose of this example is to execute a HPX-thread printing
// "Hello World!" once. That's all.

//[hello_world_1_getting_started
// Including 'hpx/hpx_main.hpp' instead of the usual 'hpx/hpx_init.hpp' enables
// to use the plain C-main below as the direct main HPX entry point.
#include <hpx/hpx_main.hpp>
#include <hpx/iostream.hpp>

int main()
{
    // Say hello to the world!
    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:

```
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.

2.5.13 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:

```
...: 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[205] 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:

```
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.

2.6 Additional material

- 2-day workshop held at CSCS in 2016
  - Recorded lectures[206]
  - Slides[207]
- Tutorials repository[208]
- STE||AR Group blog posts[209]

205 https://www.boost.org/
206 https://www.youtube.com/playlist?list=PL1tk5lGm7zvSXfS-sqOOmIJ0lFNjKze18
207 https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2016
208 https://github.com/STEllAR-GROUP/tutorials
209 http://stellar-group.org/blog/

2.7 Overview

HPX is organized into different sub-libraries. Those libraries can be seen as independent modules, with clear dependencies and no cycles. As an end-user, the use of these modules 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 *All modules* for a list of the available modules.

2.8 All modules

2.8.1 actions

TODO: High-level description of the library.

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

2.8.2 actions_base

TODO: High-level description of the library.

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

2.8.3 affinity

The affinity module contains helper functionality for mapping worker threads to hardware resources.

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

2.8.4 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.

2.8.5 allocator_support

This module provides utilities for allocators. It contains *`hpx::util::internal_allocator`* which directly forwards allocation calls to `jemalloc`. This utility is is mainly useful on Windows.

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

2.8.6 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.

2.8.7 assertion

The assertion library implements the macros *HPX_ASSERT* and *HPX_ASSERT_MSG*. Those two macros can be used to implement assertions which are turned of during a release build.

By default, the location and function where the assert has been called from are displayed when the assertion fires. This behavior can be modified by using *hpx::assertion::set_assertion_handler*. When HPX initializes, it uses this function to specify a more elaborate assertion handler. If your application needs to customize this, it needs to do so before calling `hpx::hpx_init`, `hpx::hpx_main` or using the C-main wrappers.

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

2.8.8 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 libs_async_distributed with implementations for the local and distributed cases.

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

2.8.9 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.

2.8.10 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

```
// 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
```

(continues on next page)

(continued from previous page)

```
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_CUDA_COMPUTE` 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_CUDA_COMPUTE=OFF`).

`HPX_WITH_CUDA_COMPUTE=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.

2.8.11 async

This module contains functionality for asynchronously launching work on remote localities: *`hpx::async`*, *`hpx::apply`*. This module extends the local-only functions in *async_local*.

See the API reference of this module for more details.

2.8.12 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.

2.8.13 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

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

becomes

```
hpx::future<int> f = hpx::async(executor, MPI_Isend, buf, count, datatype, rank, tag);
```

When the MPI operation is complete, the future will become ready. This allows communication to integrated cleanly with the rest of HPX, in particular the continuation style of programming may be used to build up more complex code. Consider the following example, that chains user processing, sends and receives using continuations…

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

// post an asynchronous receive using MPI_Irecv
hpx::future<int> f_recv = hpx::async(
    exec, MPI_Irecv, &data, rank, MPI_INT, rank_from, i);

// attach a continuation to run when the recv completes,
f_recv.then([=, &tokens, &counter](auto&&)
{
    // call an application specific function
    msg_recv(rank, size, rank_to, rank_from, tokens[i], i);

    // send a new message
    hpx::future<int> f_send = hpx::async(
        exec, MPI_Isend, &tokens[i], 1, MPI_INT, rank_to, i);

    // when that send completes
    f_send.then([=, &tokens, &counter](auto&&)
    {
        // call an application specific function
        msg_send(rank, size, rank_to, rank_from, tokens[i], i);
    });
}
```

The example above makes use of MPI_Isend and MPI_Irecv, but *any* MPI function that uses requests may be futurized in this manner. The following is a (non exhaustive) list of MPI functions that *should* be supported, though not all have been tested at the time of writing (please report any problems to the issue tracker).

```
int MPI_Isend(...);
int MPI_Ibsend(...);
int MPI_Issend(...);
int MPI_Irsend(...);
int MPI_Irecv(...);
int MPI_Imrecv(...);
int MPI_Ibarrier(...);
int MPI_Ibcast(...);
int MPI_Igather(...);
int MPI_Igatherv(...);
int MPI_Iscatter(...);
int MPI_Iscatterv(...);
int MPI_Iallgather(...);
int MPI_Iallgatherv(...);
int MPI_Ialltoall(...);
int MPI_Ialltoallv(...);
int MPI_Ialltoallw(...);
int MPI_Ireduce(...);
int MPI_Iallreduce(...);
int MPI_Ireduce_scatter(...);
int MPI_Ireduce_scatter_block(...);
int MPI_Iscan(...);
```

(continues on next page)

(continued from previous page)

```
int MPI_Iexscan(...);
int MPI_Ineighbor_allgather(...);
int MPI_Ineighbor_allgatherv(...);
int MPI_Ineighbor_alltoall(...);
int MPI_Ineighbor_alltoallv(...);
int MPI_Ineighbor_alltoallw(...);
```

Note that the *HPX* mpi futurization wrapper should work with *any* asynchronous *MPI* call, as long as the function signature has the last two arguments *MPI_xxx(…, MPI_Comm comm, MPI_Request *request)* - internally these two parameters will be substituted by the executor and future data parameters that are supplied by template instantiations inside the *hpx::mpi* code.

See the API reference of this module for more details.

2.8.14 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.

2.8.15 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.

2.8.16 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:

```
hpx::future<hpx::util::checkpoint> hpx::util::save_checkpoint(a, b, c, ...);
```

`save_checkpoint` takes arbitrary data containers, such as `int`, `double`, `float`, `vector`, and `future`, and serializes them into a newly created `checkpoint` object. This function returns a `future` to a `checkpoint` containing the data. Here's an example of a simple use case:

```
using hpx::util::checkpoint;
using hpx::util::save_checkpoint;

std::vector<int> vec{1,2,3,4,5};
hpx::future<checkpoint> save_checkpoint(vec);
```

Once the future is ready, the checkpoint object will contain the `vector` `vec` and its five elements.

`prepare_checkpoint` takes arbitrary data containers (same as for `save_checkpoint`), , such as `int`, `double`, `float`, `vector`, and `future`, and calculates the necessary buffer space for the checkpoint that would be created if `save_checkpoint` was called with the same arguments. This function returns a `future` to a `checkpoint` that is appropriately initialized. Here's an example of a simple use case:

```
using hpx::util::checkpoint;
using hpx::util::prepare_checkpoint;

std::vector<int> vec{1,2,3,4,5};
hpx::future<checkpoint> prepare_checkpoint(vec);
```

Once the future is ready, the checkpoint object will be initialized with an appropriately sized internal buffer.

It is also possible to modify the launch policy used by `save_checkpoint`. This is accomplished by passing a launch policy as the first argument. It is important to note that passing `hpx::launch::sync` will cause `save_checkpoint` to return a `checkpoint` instead of a `future` to a `checkpoint`. All other policies passed to `save_checkpoint` will return a `future` to a `checkpoint`.

Sometimes `checkpoint` s must be declared before they are used. `save_checkpoint` allows users to move pre-created `checkpoint` s into the function as long as they are the first container passing into the function (In the case where a launch policy is used, the `checkpoint` will immediately follow the launch policy). An example of these features can be found below:

```
    char character = 'd';
    int integer = 10;
    float flt = 10.01f;
    bool boolean = true;
    std::string str = "I am a string of characters";
    std::vector<char> vec(str.begin(), str.end());
    checkpoint archive;

    // Test 1
    //  test basic functionality
    hpx::shared_future<checkpoint> f_archive = save_checkpoint(
        std::move(archive), character, integer, flt, boolean, str, vec);
```

Once users can create `checkpoints` they must now be able to restore the objects they contain into memory. This is accomplished by the function `restore_checkpoint`. This function takes a `checkpoint` and fills its data into the containers it is provided. It is important to remember that the containers must be ordered in the same way they were placed into the `checkpoint`. For clarity see the example below:

```
    char character2;
    int integer2;
    float flt2;
    bool boolean2;
    std::string str2;
    std::vector<char> vec2;

    restore_checkpoint(data, character2, integer2, flt2, boolean2, str2, vec2);
```

The core utility of `checkpoint` is in its ability to make certain data persistent. Often, this means that the data needs to be stored in an object, such as a file, for later use. *HPX* has two solutions for these issues: stream operator overloads and access iterators.

HPX contains two stream overloads, `operator<<` and `operator>>`, to stream data out of and into `checkpoint`. Here is an example of the overloads in use below:

```
double a9 = 1.0, b9 = 1.1, c9 = 1.2;
std::ofstream test_file_9("test_file_9.txt");
hpx::future<checkpoint> f_9 = save_checkpoint(a9, b9, c9);
test_file_9 << f_9.get();
test_file_9.close();

double a9_1, b9_1, c9_1;
std::ifstream test_file_9_1("test_file_9.txt");
checkpoint archive9;
test_file_9_1 >> archive9;
restore_checkpoint(archive9, a9_1, b9_1, c9_1);
```

This is the primary way to move data into and out of a `checkpoint`. It is important to note, however, that users should be cautious when using a stream operator to load data and another function to remove it (or vice versa). Both `operator<<` and `operator>>` rely on a `.write()` and a `.read()` function respectively. In order to know how much data to read from the `std::istream`, the `operator<<` will write the size of the `checkpoint` before writing the `checkpoint` data. Correspondingly, the `operator>>` will read the size of the stored data before reading the data into a new instance of `checkpoint`. As long as the user employs the `operator<<` and `operator>>` to stream the data, this detail can be ignored.

Important: Be careful when mixing `operator<<` and `operator>>` with other facilities to read and write to a `checkpoint`. `operator<<` writes an extra variable, and `operator>>` reads this variable back separately. Used together the user will not encounter any issues and can safely ignore this detail.

Users may also move the data into and out of a `checkpoint` using the exposed `.begin()` and `.end()` iterators. An example of this use case is illustrated below.

```
std::ofstream test_file_7("checkpoint_test_file.txt");
std::vector<float> vec7{1.02f, 1.03f, 1.04f, 1.05f};
hpx::future<checkpoint> fut_7 = save_checkpoint(vec7);
checkpoint archive7 = fut_7.get();
std::copy(archive7.begin(),     // Write data to ofstream
    archive7.end(),             // ie. the file
    std::ostream_iterator<char>(test_file_7));
test_file_7.close();

std::vector<float> vec7_1;
std::vector<char> char_vec;
std::ifstream test_file_7_1("checkpoint_test_file.txt");
if (test_file_7_1)
{
    test_file_7_1.seekg(0, test_file_7_1.end);
    auto length = test_file_7_1.tellg();
    test_file_7_1.seekg(0, test_file_7_1.beg);
    char_vec.resize(length);
    test_file_7_1.read(char_vec.data(), length);
}
checkpoint archive7_1(std::move(char_vec));    // Write data to checkpoint
restore_checkpoint(archive7_1, vec7_1);
```

Checkpointing components

save_checkpoint and restore_checkpoint are also able to store components inside checkpoints. This can be done in one of two ways. First a client of the component can be passed to save_checkpoint. When the user wishes to resurrect the component she can pass a client instance to restore_checkpoint.

This technique is demonstrated below:

```
// Try to checkpoint and restore a component with a client
std::vector<int> vec3{10, 10, 10, 10, 10};

// Create a component instance through client constructor
data_client D(hpx::find_here(), std::move(vec3));
hpx::future<checkpoint> f3 = save_checkpoint(D);

// Create a new client
data_client E;

// Restore server inside client instance
restore_checkpoint(f3.get(), E);
```

The second way a user can save a component is by passing a shared_ptr to the component to save_checkpoint. This component can be resurrected by creating a new instance of the component type and passing a shared_ptr to the new instance to restore_checkpoint.

This technique is demonstrated below:

```
// test checkpoint a component using a shared_ptr
std::vector<int> vec{1, 2, 3, 4, 5};
data_client A(hpx::find_here(), std::move(vec));

// Checkpoint Server
hpx::id_type old_id = A.get_id();

hpx::future<std::shared_ptr<data_server>> f_a_ptr =
    hpx::get_ptr<data_server>(A.get_id());
std::shared_ptr<data_server> a_ptr = f_a_ptr.get();
hpx::future<checkpoint> f = save_checkpoint(a_ptr);
auto&& data = f.get();

// test prepare_checkpoint API
checkpoint c = prepare_checkpoint(hpx::launch::sync, a_ptr);
HPX_TEST(c.size() == data.size());

// Restore Server
// Create a new server instance
std::shared_ptr<data_server> b_server;
restore_checkpoint(data, b_server);
```

2.8.17 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.

2.8.18 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_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::lcos::barrier`*: distributed barrier.
- `hpx::lcos::broadcast`: performs a given action on all given global identifiers.
- `hpx::lcos::fold`: performs a fold with a given action on all given global identifiers.
- `hpx::lcos::gather`: gathers values from all participating sites.
- *`hpx::lcos::latch`*: distributed latch.
- `hpx::lcos::reduce`: performs a reduction on data from each participating site to a root site.
- *`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.

2.8.19 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.

2.8.20 components_base

TODO: High-level description of the library.

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

2.8.21 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.

2.8.22 compute_cuda

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

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

2.8.23 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.

2.8.24 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.

2.8.25 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.

2.8.26 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.

2.8.27 coroutines

The coroutines module provides coroutine (user-space thread) implementations for different platforms.

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

2.8.28 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.

2.8.29 debugging

This module provides helpers for demangling symbol names.

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

2.8.30 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.

2.8.31 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.

2.8.32 execution_base

The basic execution module is the main entry point to implement parallel and concurrent operations. It is modeled after P0443[210] 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.

[210] http://wg21.link/p0443

- **`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.

2.8.33 executors

The executors module exposes executors and execution policies. Most importantly, it exposes the following classes and constants:

- *`hpx::parallel::execution::sequenced_executor`*
- *`hpx::parallel::execution::parallel_executor`*
- *`hpx::parallel::execution::sequenced_policy`*
- *`hpx::parallel::execution::parallel_policy`*
- *`hpx::parallel::execution::parallel_unsequenced_policy`*
- *`hpx::parallel::execution::sequenced_task_policy`*
- *`hpx::parallel::execution::parallel_task_policy`*
- `hpx::parallel::execution::seq`
- `hpx::parallel::execution::par`
- `hpx::parallel::execution::par_unseq`
- `hpx::parallel::execution::task`

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

2.8.34 executors_distributed

This module provides the executor `hpx::parallel::execution::disribution_policy_executor`. It allows one to create work that is implicitly distributed over multiple localities.

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

2.8.35 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.

2.8.36 format

The format module exposes the `format` and `format_to` functions for formatting strings.

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

2.8.37 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.

2.8.38 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.

2.8.39 hardware

The hardware module abstracts away hardware specific details of timestamps and CPU features.

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

2.8.40 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.

2.8.41 include

This module provides no functionality in itself. Instead it provides headers that group together other headers that often appear together. The grouping is similar to that provided by modules. We intend to deprecate these headers in favor of *hpx/<modulename>.hpp* headers.

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

2.8.42 init_runtime

TODO: High-level description of the library.

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

2.8.43 io_service

This module provides an abstraction over Boost.ASIO, combining multiple `boost::asio::io_services` into a single pool. *hpx::util::io_service_pool* provides a simple pool of `boost::asio::io_services` with an API similar to `boost::asio::io_service`. `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.

2.8.44 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.

2.8.45 itt_notify

This module provides support for profiling with Intel VTune[211].

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

[211] https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html

2.8.46 lcos_distributed

This module contains distributed *LCO*s. Currently the only LCO provided is :cpp:class::*hpx::lcos::channel*, a construct for sending values from one *locality* to another. See *lcos_local* for local LCOs.

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

2.8.47 lcos_local

This module provides the following local *LCO*s:

- *`hpx::lcos::local::and_gate`*
- `hpx::lcos::local::channel`
- `hpx::lcos::local::one_element_channel`
- `hpx::lcos::local::receive_channel`
- `hpx::lcos::local::send_channel`
- *`hpx::lcos::local::guard`*
- *`hpx::lcos::local::guard_set`*
- *`hpx::lcos::local::run_guarded`*
- *`hpx::lcos::local::conditional_trigger`*
- `hpx::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.

2.8.48 logging

This module provides useful macros for logging information.

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

2.8.49 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.

2.8.50 mpi_base

This module provides helper functionality for detecting MPI environments.

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

2.8.51 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.

2.8.52 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.

2.8.53 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.

2.8.54 plugin

This module provides base utilities for creating plugins.

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

2.8.55 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.

2.8.56 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.

2.8.57 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[212]. 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.

2.8.58 resiliency

In *HPX*, a program failure is a manifestation of a failing task. This module exposes several APIs that allow users to manage failing tasks in a convenient way by either replaying a failed task or by replicating a specific task.

Task replay is analogous to the Checkpoint/Restart mechanism found in conventional execution models. The key difference being localized fault detection. When the runtime detects an error, it replays the failing task as opposed to completely rolling back the entire program to the previous checkpoint.

Task replication is designed to provide reliability enhancements by replicating a set of tasks and evaluating their results to determine a consensus among them. This technique is most effective in situations where there are few tasks in the critical path of the DAG which leaves the system underutilized or where hardware or software failures may result in an incorrect result instead of an error. However, the drawback of this method is the additional computational cost incurred by repeating a task multiple times.

The following API functions are exposed:

- *`hpx::resiliency::experimental::async_replay`*: This version of task replay will catch user-defined exceptions and automatically reschedule the task N times before throwing an `hpx::resiliency::experimental::abort_replay_exception` if no task is able to complete execution without an exception.
- *`hpx::resiliency::experimental::async_replay_validate`*: This version of replay adds an argument to async replay which receives a user-provided validation function to test the result of the task against. If the task's output is validated, the result is returned. If the output fails the check or an exception is thrown, the task is replayed until no errors are encountered or the number of specified retries has been exceeded.
- *`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

[212] https://www.boost.org/doc/libs/1_70_0/doc/html/program_options.html

user to provide a custom consensus function to properly form a consensus. This voting function then returns the “correct”” answer.

- *hpx::resiliency::experimental::async_replicate_vote_validate*: This combines the features of the previously discussed replicate set. Replicate vote validate allows a user to provide a validation function to filter results. Additionally, as described in replicate vote, the user can provide a “voting function” which returns the consensus formed by the voting logic.
- *hpx::resiliency::experimental::dataflow_replay*: This version of dataflow replay will catch user-defined exceptions and automatically reschedules the task N times before throwing an `hpx::resiliency::experimental::abort_replay_exception` if no task is able to complete execution without an exception. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replay_validate*: This version of replay adds an argument to dataflow replay which receives a user-provided validation function to test the result of the task against. If the task’s output is validated, the result is returned. If the output fails the check or an exception is thrown, the task is replayed until no errors are encountered or the number of specified retries have been exceeded. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate*: This is the most basic implementation of the task replication. The API returns the first result that runs without detecting any errors. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate_validate*: This API additionally takes a validation function which evaluates the return values produced by the threads. The first task to compute a valid result is returned. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate_vote*: This API adds a vote function to the basic replicate function. Many hardware or software failures are silent errors which do not interrupt program flow. In order to detect errors of this kind, it is necessary to run the task several times and compare the values returned by every version of the task. In order to determine which return value is “correct”, the API allows the user to provide a custom consensus function to properly form a consensus. This voting function then returns the “correct” answer. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.
- *hpx::resiliency::experimental::dataflow_replicate_vote_validate*: This combines the features of the previously discussed replicate set. Replicate vote validate allows a user to provide a validation function to filter results. Additionally, as described in replicate vote, the user can provide a “voting function” which returns the consensus formed by the voting logic. Any arguments for the executed task that are futures will cause the task invocation to be delayed until all of those futures have become ready.

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

2.8.59 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.

2.8.60 runtime_configuration

This module handles the configuration options required by the runtime.

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

2.8.61 runtime_local

TODO: High-level description of the library.

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

2.8.62 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.

2.8.63 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.

2.8.64 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[213] 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.

[213] https://www.boost.org/doc/libs/1_72_0/libs/serialization/doc/index.html

2.8.65 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.

2.8.66 statistics

This module provide some statistics utilities like rolling min/max and histogram.

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

2.8.67 string_util

This module contains string utilities inspired by the Boost string algorithms library.

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

2.8.68 synchronization

This module provides synchronization primitives which should be used rather than the C++ standard ones in *HPX* threads:

- `hpx::lcos::local::barrier`
- `hpx::lcos::local::condition_variable`
- `hpx::lcos::local::counting_semaphore`
- `hpx::lcos::local::event`
- `hpx::lcos::local::latch`
- `hpx::lcos::local::mutex`
- `hpx::lcos::local::no_mutex`
- `hpx::lcos::local::once_flag`
- `hpx::lcos::local::recursive_mutex`
- `hpx::lcos::local::shared_mutex`
- `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.

2.8.69 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.

2.8.70 thread_executors

This module provides executors implementing the executor interface proposed in N3562[214]. These executors are deprecated.

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

2.8.71 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.

[214] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf

2.8.72 thread_support

This module provides miscellaneous utilities for threading and concurrency.

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

2.8.73 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.

2.8.74 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.

2.8.75 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.

2.8.76 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.

2.8.77 timing

This module provides the timing utilities (clocks and timers).

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

2.8.78 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)[215] library. The hpx::threads::cpu_mask is a small companion class that represents a set of resources on a node.

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

2.8.79 type_support

This module provides helper facilities related to types.

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

2.8.80 util

The util module provides miscellaneous standalone utilities.

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

2.8.81 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.

2.9 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.

[215] https://www.open-mpi.org/projects/hwloc/

2.9.1 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`

Corresponds to the C++ standard library header algorithm[216]. See *Using parallel algorithms* for more information about the parallel algorithms.

Classes

- *hpx::parallel::v2::reduction*
- *hpx::parallel::v2::induction*

Functions

- *hpx::parallel::v1::adjacent_find*
- *hpx::all_of*
- *hpx::any_of*
- *hpx::copy*
- *hpx::copy_if*
- *hpx::copy_n*
- *hpx::count*
- *hpx::count_if*
- *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::parallel::v1::includes*
- *hpx::parallel::v1::inplace_merge*

[216] http://en.cppreference.com/w/cpp/header/algorithm

- *hpx::parallel::v1::is_heap*
- *hpx::parallel::v1::is_heap_until*
- *hpx::parallel::v1::is_partitioned*
- *hpx::parallel::v1::is_sorted*
- *hpx::parallel::v1::is_sorted_until*
- *hpx::parallel::v1::lexicographical_compare*
- *hpx::parallel::v1::max_element*
- *hpx::parallel::v1::merge*
- *hpx::parallel::v1::min_element*
- *hpx::parallel::v1::minmax_element*
- hpx::parallel::v1::mismatch
- *hpx::move*
- *hpx::none_of*
- *hpx::parallel::v1::partition*
- *hpx::parallel::v1::partition_copy*
- *hpx::parallel::v1::remove*
- *hpx::parallel::v1::remove_copy*
- *hpx::parallel::v1::remove_copy_if*
- *hpx::parallel::v1::remove_if*
- *hpx::parallel::v1::replace*
- *hpx::parallel::v1::replace_copy*
- *hpx::parallel::v1::replace_copy_if*
- *hpx::parallel::v1::replace_if*
- *hpx::parallel::v1::reverse*
- *hpx::parallel::v1::reverse_copy*
- *hpx::parallel::v1::rotate*
- *hpx::parallel::v1::rotate_copy*
- *hpx::parallel::v1::search*
- *hpx::parallel::v1::search_n*
- *hpx::parallel::v1::set_difference*
- *hpx::parallel::v1::set_intersection*
- *hpx::parallel::v1::set_symmetric_difference*
- *hpx::parallel::v1::set_union*
- *hpx::parallel::v1::sort*
- *hpx::parallel::v1::stable_partition*
- *hpx::parallel::v1::stable_sort*

- *hpx::parallel::v1::swap_ranges*
- *hpx::parallel::v1::unique*
- *hpx::parallel::v1::unique_copy*
- *hpx::for_loop*
- *hpx::for_loop_strided*
- *hpx::for_loop_n*
- *hpx::for_loop_n_strided*
- *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::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::move`
- *hpx::ranges::none_of*

Header `hpx/any.hpp`

Corresponds to the C++ standard library header any[217]. *hpx::util::any* is compatible with `std::any`.

[217] http://en.cppreference.com/w/cpp/header/any

Classes

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

Functions

- *hpx::util::make_any*
- *hpx::util::make_any_nonser*
- *hpx::util::make_unique_any_nonser*

Header `hpx/assert.hpp`

Corresponds to the C++ standard library header cassert[218]. *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 *Header hpx/distributed/barrier.hpp*.

Header `hpx/local/barrier.hpp`

Corresponds to the C++ standard library header barrier[219].

Classes

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

[218] http://en.cppreference.com/w/cpp/header/cassert
[219] http://en.cppreference.com/w/cpp/header/barrier

Header `hpx/distributed/barrier.hpp`

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/channel.hpp`

This header includes *Header hpx/local/channel.hpp* and *Header hpx/distributed/channel.hpp*.

Header `hpx/local/channel.hpp`

Contains a local channel implementation.

Classes

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

Header `hpx/distributed/channel.hpp`

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/chrono.hpp`

Corresponds to the C++ standard library header chrono[220]. The following replacements and extensions are provided compared to chrono[221]. The classes below are also available in the `hpx::chrono` namespace, not in the top-level `hpx` namespace.

[220] http://en.cppreference.com/w/cpp/header/chrono
[221] http://en.cppreference.com/w/cpp/header/chrono

Classes

- *hpx::util::high_resolution_clock*
- *hpx::util::high_resolution_timer*
- *hpx::util::steady_time_point*

Header `hpx/condition_variable.hpp`

Corresponds to the C++ standard library header condition_variable[222].

Classes

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

Header `hpx/exception.hpp`

Corresponds to the C++ standard library header exception[223]. *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`

Corresponds to the C++ standard library header execution[224]. 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 also available in the `hpx::execution` namespace, but not in the top-level `hpx` namespace.

[222] http://en.cppreference.com/w/cpp/header/condition_variable
[223] http://en.cppreference.com/w/cpp/header/exception
[224] http://en.cppreference.com/w/cpp/header/execution

Constants

- *hpx::parallel::execution::seq*
- *hpx::parallel::execution::par*
- *hpx::parallel::execution::par_unseq*
- *hpx::parallel::execution::task*

Classes

- *hpx::parallel::execution::sequenced_policy*
- *hpx::parallel::execution::parallel_policy*
- *hpx::parallel::execution::parallel_unsequenced_policy*
- *hpx::parallel::execution::sequenced_task_policy*
- *hpx::parallel::execution::parallel_task_policy*
- *hpx::parallel::execution::auto_chunk_size*
- *hpx::parallel::execution::dynamic_chunk_size*
- *hpx::parallel::execution::guided_chunk_size*
- *hpx::parallel::execution::persistent_auto_chunk_size*
- *hpx::parallel::execution::static_chunk_size*

Header `hpx/functional.hpp`

Corresponds to the C++ standard library header functional[225]. `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*

[225] http://en.cppreference.com/w/cpp/header/functional

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 *Header hpx/local/future.hpp* and *Header hpx/distributed/future.hpp*.

Header `hpx/local/future.hpp`

Corresponds to the C++ standard library header future[226]. 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*

[226] http://en.cppreference.com/w/cpp/header/future

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)

(continued from previous 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 `hpx/distributed/future.hpp`

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/init.hpp`

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

Classes

- *hpx::init_params*
- *hpx::runtime_mode*

Functions

- *hpx::init*
- *hpx::start*
- *hpx::finalize*
- *hpx::disconnect*
- *hpx::suspend*
- *hpx::resume*

Header `hpx/latch.hpp`

This header includes *Header hpx/local/latch.hpp* and *Header hpx/distributed/latch.hpp*.

Header `hpx/local/latch.hpp`

Corresponds to the C++ standard library header latch[227].

Classes

- *`hpx::lcos::local::cpp20_latch`*

Header `hpx/distributed/latch.hpp`

Contains a distributed latch implementation. This functionality is also exposed through the `hpx::distributed` namespace. The name in `hpx::distributed` should be preferred.

Classes

- *`hpx::lcos::latch`*

Header `hpx/mutex.hpp`

Corresponds to the C++ standard library header mutex[228].

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`

[227] http://en.cppreference.com/w/cpp/header/latch
[228] http://en.cppreference.com/w/cpp/header/mutex

Functions

- *hpx::lcos::local::call_once*

Header `hpx/memory.hpp`

Corresponds to the C++ standard library header memory[229]. It contains parallel versions of the copy, fill, move, and construct helper functions in memory[230]. See *Using parallel algorithms* for more information about the parallel algorithms.

Functions

- *hpx::parallel::v1::uninitialized_copy*
- *hpx::parallel::v1::uninitialized_copy_n*
- *hpx::parallel::v1::uninitialized_default_construct*
- *hpx::parallel::v1::uninitialized_default_construct_n*
- *hpx::parallel::v1::uninitialized_fill*
- *hpx::parallel::v1::uninitialized_fill_n*
- *hpx::parallel::v1::uninitialized_move*
- *hpx::parallel::v1::uninitialized_move_n*
- *hpx::parallel::v1::uninitialized_value_construct*
- *hpx::parallel::v1::uninitialized_value_construct_n*

Header `hpx/numeric.hpp`

Corresponds to the C++ standard library header numeric[231]. See *Using parallel algorithms* for more information about the parallel algorithms.

Functions

- *hpx::parallel::v1::adjacent_difference*
- *hpx::parallel::v1::exclusive_scan*
- *hpx::parallel::v1::inclusive_scan*
- *hpx::reduce*
- *hpx::parallel::v1::transform_exclusive_scan*
- *hpx::parallel::v1::transform_inclusive_scan*
- *hpx::transform_reduce*

[229] http://en.cppreference.com/w/cpp/header/memory
[230] http://en.cppreference.com/w/cpp/header/memory
[231] http://en.cppreference.com/w/cpp/header/numeric

Header `hpx/optional.hpp`

Corresponds to the C++ standard library header optional[232]. `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 *Header hpx/distributed/runtime.hpp*.

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*

[232] http://en.cppreference.com/w/cpp/header/optional

- *hpx::get_num_localities*
- *hpx::get_locality_name*

Header `hpx/distributed/runtime.hpp`

This header 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/system_error.hpp`

Corresponds to the C++ standard library header system_error[233].

Classes

- *hpx::error_code*

Header `hpx/task_block.hpp`

Corresponds to the `task_block` feature in N4411[234]. See *Using task blocks* for more details on using task blocks.

Classes

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

[233] http://en.cppreference.com/w/cpp/header/system_error
[234] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf

Functions

- *hpx::parallel::v2::define_task_block*
- *hpx::parallel::v2::define_task_block_restore_thread*

Header `hpx/thread.hpp`

Corresponds to the C++ standard library header thread[235]. 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*

Header `hpx/semaphore.hpp`

Corresponds to the C++ standard library header semaphore[236].

Classes

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

Header `hpx/shared_mutex.hpp`

Corresponds to the C++ standard library header shared_mutex[237].

[235] http://en.cppreference.com/w/cpp/header/thread
[236] http://en.cppreference.com/w/cpp/header/semaphore
[237] http://en.cppreference.com/w/cpp/header/shared_mutex

Classes

- *hpx::lcos::local::shared_mutex*

Header hpx/stop_token.hpp

Corresponds to the C++ standard library header stop_token[238].

Constants

- hpx::nostopstate

Classes

- hpx::stop_callback
- *hpx::stop_source*
- *hpx::stop_token*
- *hpx::nostopstate_t*

Header hpx/tuple.hpp

Corresponds to the C++ standard library header tuple[239]. hpx::util::tuple can be used in CUDA device code, unlike std::tuple.

Constants

- *hpx::util::ignore*

Classes

- hpx::util::tuple
- *hpx::util::tuple_size*
- hpx::util::tuple_element

[238] http://en.cppreference.com/w/cpp/header/stop_token
[239] http://en.cppreference.com/w/cpp/header/tuple

Functions

- *hpx::util::make_tuple*
- *hpx::util::tie*
- *hpx::util::forward_as_tuple*
- *hpx::util::tuple_cat*
- hpx::util::get

Header hpx/type_traits.hpp

Corresponds to the C++ standard library header type_traits[240]. Provides `hpx::util::invoke_result` as a replacement for `std::invoke_result`.

Classes

- hpx::util::invoke_result

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*

[240] http://en.cppreference.com/w/cpp/header/type_traits

- *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.

2.9.2 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.

template<typename **Action**>
struct async_result

#include <colocating_distribution_policy.hpp>

Note This function is part of the invocation policy implemented by this class

Public Types

template<>
using type = *hpx*::future<**typename** *traits*::promise_local_result<**typename** *hpx*::*traits*::*extract_action*<Action>::remote_

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>::remote_

struct binpacking_distribution_policy

#include <binpacking_distribution_policy.hpp> This class specifies the parameters for a binpacking distribution policy to use for creating a given number of items on a given set of localities. The binpacking policy will distribute the new objects in a way such that each of the localities will equalize the number of overall objects of this type based on a given criteria (by default this criteria is the overall number of objects of this type).

Public Functions

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 which 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

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 os current located

Parameters

- `id`: [in] The global address of the object with which the new instances should be colocated on

template<typename **Client**, typename **Stub**>
colocating_distribution_policy **operator()**(client_base<*Client*, *Stub*> **const** &*client*) **const**
Create a new colocating_distribution_policy representing the locality where the given object os current located

Parameters

- `client`: [in] The client side representation of the object with which the new instances should be colocated on

template<typename **Component**, typename ...**Ts**>
hpx::future<*hpx*::id_type> **create**(*Ts*&&... *vs*) **const**
Create one object on the locality of the object this distribution policy instance is associated with

Note This function is part of the placement policy implemented by this class

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 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.

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

template<typename **Action**, typename **Continuation**, typename ...**Ts**>
bool **apply** (*Continuation* &&*c*, *threads*::*thread_priority priority*, *Ts*&&... *vs*) **const**

Note This function is part of the invocation policy implemented by this class

template<typename **Action**, typename ...**Ts**>
bool **apply** (*threads*::*thread_priority priority*, *Ts*&&... *vs*) **const**

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
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

template<typename **Action**, typename **Callback**, typename ...**Ts**>
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

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

default_distribution_policy()
Default-construct a new instance of a default_distribution_policy. This policy will represent one locality (the local locality).

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

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

default_distribution_policy **operator()**(id_type **const** &*loc*) **const**
Create a new *default_distribution* policy representing the given locality

Parameters

- `loc`: [in] The locality the new instance should represent

template<typename **Component**, typename ...**Ts**>
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.

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

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.

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

template<typename **Action**, typename **Continuation**, typename ...**Ts**>
bool **apply**(*Continuation &&c*, *threads::thread_priority priority*, *Ts&&... vs*) **const**

Note This function is part of the invocation policy implemented by this class

template<typename **Action**, typename ...**Ts**>
bool **apply**(*threads::thread_priority priority*, *Ts&&... vs*) **const**

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
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

template<typename **Action**, typename **Callback**, typename ...**Ts**>
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

namespace hpx

Functions

std::vector<Client> **find_all_from_basename**(*std*::string *base_name*, *std*::size_t *num_ids*)
Return all registered ids from all localities from the given base name.

This function locates all ids which were registered with the given base name. It returns a list of futures representing those ids.

Return all registered clients 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 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.

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 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.

std::vector<Client> **find_from_basename**(*std*::string *base_name*, *std*::vector<*std*::size_t> **const** &*ids*)

Return registered ids from the given base name and sequence numbers.

This function locates the ids which were registered with the given base name and the given sequence numbers. It returns a list of futures representing those ids.

Return registered clients 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 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.

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 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.

Client **find_from_basename**(*std*::string *base_name*, *std*::size_t *sequence_nr* = ~static_cast<*std*::size_t>(0))

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 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 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.

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.

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.

template<typename **Client**, typename **Stub**>
hpx::future<bool> **register_with_basename**(*std*::string *base_name*, *components*::client_base<*Client*, *Stub*> &*client*, *std*::size_t *sequence_nr* = ~static_cast<*std*::size_t>(0))

Register the id wrapped in the given client using the given base name.

The function registers the object the given client refers to using the provided base name.

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.

Client **unregister_with_basename**(*std*::string *base_name*, *std*::size_t *sequence_nr* = ~static_cast<*std*::size_t>(0))

Unregister the given id using the given base name.

The function unregisters the given ids using the provided base name.

Unregister the given base name.

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.

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.

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.

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.

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`**(*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.

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 *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 remote localities.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

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 *hpx::naming::invalid_id*.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

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_here(), hpx::find_all_localities()

Parameters

- `type`: [in] The type of the components for which the function should return any available locality.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

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.

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.

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.

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

- `The`: only template parameter has to be the type of the server side component.

template<typename **Derived**, typename **Stub**>
hpx::future<*std*::shared_ptr<**typename** *components*::client_base<*Derived*, *Stub*>::server_component_type>> **get_ptr**(*compo* *Stub*> **cons** &*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.

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.

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.
- `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.

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.

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)*

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

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*.

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*.

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*.

template<typename **Result**>
std::enable_if<!*std*::is_same<**typename** *util*::decay<*Result*>::type, *naming*::address>::value>::type **set_lco_value**(*naming* **const** &*id*, *Result* &&*t*, bool *move_* = true)

Set the result value for the (managed) LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the given value.
- `t`: [in] This is the value which should be sent to the LCO.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
std::enable_if<!*std*::is_same<**typename** *util*::decay<*Result*>::type, *naming*::address>::value>::type **set_lco_value_unman**

Set the result value for the (unmanaged) LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the given value.
- `t`: [in] This is the value which should be sent to the LCO.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
void **set_lco_value**(*naming*::id_type **const** &*id*, *naming*::address &&*addr*, *Result* &&*t*, *naming*::id_type **const** &*cont*, bool *move_credits* = true)

Set the result value for the LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the given value.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `t`: [in] This is the value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
std::enable_if<!*std*::is_same<**typename** *util*::decay<*Result*>::type, *naming*::address>::value>::type **set_lco_value**(*naming* **const** &*id*, *Result* &&*t*, *naming*::id_ **const** &*cont*, bool *move_c* = true)

Set the result value for the (managed) LCO referenced by the given id.

Parameters

- `id`: [in] This represents the id of the LCO which should receive the given value.
- `t`: [in] This is the value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

template<typename **Result**>
std::enable_if<!*std*::is_same<**typename** *util*::decay<*Result*>::type, *naming*::address>::value>::type **set_lco_value_unman**

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*, *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.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `e`: [in] This is the error value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*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.
- `addr`: [in] This represents the addr of the LCO which should be triggered.
- `e`: [in] This is the error value which should be sent to the LCO.
- `cont`: [in] This represents the LCO to trigger after completion.
- `move_credits`: [in] If this is set to *true* then it is ok to send all credits in *id* along with the generated message. The default value is *true*.

void **set_lco_error**(*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*.

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.

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*.

`template<typename Component, typename ... Ts><unspecified> hpx::new_(id_type const & l`

Create one or more new instances of the given Component type on the specified locality.

This function creates one or more new instances of the given Component type on the specified locality and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
   hpx::new_<some_component>(hpx::find_here(), ...);
hpx::id_type id = f.get();
```

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.

`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:

```
hpx::future<hpx::id_type> f =
   hpx::local_new<some_component>(...);
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 first argument is `hpx::launch::sync` the function will directly return an `hpx::id_type`.
- If the explicit template argument *Component* represents a client side object (`traits::is_client<Component>::value` evaluates to true), the function will

return a new instance of that type which can be used to refer to the newly created component instance.

Note The difference of this function to hpx::new_ is that it can be used in cases where the supplied arguments are non-copyable and non-movable. All operations are guaranteed to be local only.

Parameters

- `vs`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

template<typename Component, typename ... Ts><unspecified> hpx::new_(id_type const & lo

Create multiple new instances of the given Component type on the specified locality.

This function creates multiple new instances of the given Component type on the specified locality and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type> > f =
   hpx::new_<some_component[]>(hpx::find_here(), 10, ...);
hpx::id_type id = f.get();
```

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

- `locality`: [in] The global address of the locality where the new instance should be created on.
- `count`: [in] The number of component instances to create
- `vs`: [in] Any number of arbitrary arguments (passed by value, by const reference or by rvalue reference) which will be forwarded to the constructor of the created component instance.

template<typename Component, typename DistPolicy, typename ... Ts><unspecified> hpx::ne

Create one or more new instances of the given Component type based on the given distribution policy.

This function creates one or more new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for global address which can be used to reference the new component instance(s).

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<hpx::id_type> f =
   hpx::new_<some_component>(hpx::default_layout, ...);
hpx::id_type id = f.get();
```

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.

template<typename Component, typename DistPolicy, typename ... Ts><unspecified> hpx::n

Create multiple new instances of the given Component type on the localities as defined by the given distribution policy.

This function creates multiple new instances of the given Component type on the localities defined by the given distribution policy and returns a future object for the global address which can be used to reference the new component instance.

Note This function requires to specify an explicit template argument which will define what type of component(s) to create, for instance:

```
hpx::future<std::vector<hpx::id_type> > f =
   hpx::new_<some_component[]>(hpx::default_layout, 10, ...);
hpx::id_type id = f.get();
```

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.

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.

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 *migrate_to_storage* 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 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_storage`: [in] The id of the storage facility to migrate this object to.

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.

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.

Variables

char **const** ***const default_binpacking_counter_name** = "/runtime{locality/total}/count/component@"

binpacking_distribution_policy **const binpacked**

A predefined instance of the binpacking *distribution_policy*. It will represent the local locality and will place all items to create here.

colocating_distribution_policy **const colocated**

A predefined instance of the co-locating *distribution_policy*. It will represent the local locality and will place all items to create here.

default_distribution_policy **const default_layout** = {}

A predefined instance of the default *distribution_policy*. It will represent the local locality and will place all items to create here.

namespace naming

Functions

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.

file **migrate_from_storage.hpp**

#include <hpx/config.hpp>#include <hpx/components_base/traits/is_component.hpp>#include <hpx/futures/future.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/modules/memory.hpp>#include <hpx/runtime/naming_fwd.hpp>#include <hpx/modules/naming_base.hpp>#include <hpx/runtime/agas_fwd.hpp>#include <hpx/functional/function.hpp>#include <hpx/runtime_configuration/agas_service_mode.hpp>#include <cstdint>#include <string>#include <hpx/serialization/serialization_fwd.hpp>#include <iosfwd>#include <utility>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/components/component_storage/server/migrate_from_storage.hpp>#include <type_traits>

file **migrate_to_storage.hpp**

#include <hpx/config.hpp>#include <hpx/components_base/traits/is_component.hpp>#include <hpx/futures/future.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/actions_base/traits/action_remote_result.hpp>#include <hpx/assert.hpp>#include <hpx/functional/bind_back.hpp>#include <hpx/futures/traits/acquire_future.hpp>#include <hpx/futures/traits/future_access.hpp>#include <hpx/futures/traits/future_traits.hpp>#include <hpx/futures/traits/is_future.hpp>#include <hpx/memory/intrusive_ptr.hpp>#include <hpx/modules/errors.hpp>#include <hpx/modules/memory.hpp>#include <hpx/runtime/agas/interface.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/functional/unique_function.hpp>#include <hpx/preprocessor/cat.hpp>#include <hpx/preprocessor/expand.hpp>#include <hpx/preprocessor/nargs.hpp>#include <hpx/preprocessor/stringize.hpp>#include <hpx/preprocessor/strip_parens.hpp>#include <hpx/runtime/naming_fwd.hpp>#include <hpx/thread_support/atomic_count.hpp>#include <hpx/traits/component_type_database.hpp>#include <cstdint>#include <hpx/type_support/decay.hpp>#include <string>#include <hpx/runtime/components_fwd.hpp>#include <hpx/traits/managed_component_policies.hpp>#include <hpx/type_support/always_void.hpp>#include <cstddef>#include <hpx/async_base/launch_policy.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/allocator_support/internal_allocator.hpp>#include <hpx/execution_base/register_locks.hpp>#include <hpx/execution_base/this_thread.hpp>#include <hpx/concurrency/spinlock_pool.hpp>#include <hpx/futures/traits/get_remote_result.hpp>#include <hpx/futures/traits/promise_local_result.hpp>#include <hpx/modules/itt_notify.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/serialization/traits/is_bitwise_serializable.hpp>#include <functional>#include <iosfwd>#include <mutex>#include <vector>#include <hpx/config/warnings_prefix.hpp>#include <hpx/runtime/naming/id_type_impl.hpp>#include <hpx/config/warnings_suffix.hpp>#include <boost/dynamic_bitset.hpp>#include <map>#include <utility>#include <hpx/runtime/components/make_client.hpp>#include <hpx/traits/is_client.hpp>#include <type_traits>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/async_distributed/detail/async_colocated_fwd.hpp>#include <hpx/async_distributed/detail/async_implementations_fwd.hp <hpx/async_local/async_fwd.hpp>#include <hpx/runtime/naming/unmanaged.hpp>#include

<hpx/serialization/serialize.hpp>#include <exception>#include <hpx/components/component_storage/component_storage.hpp> <hpx/components/component_storage/server/migrate_to_storage.hpp>

file **basename_registration_fwd.hpp**

#include <hpx/config.hpp>#include <hpx/components_fwd.hpp>#include <hpx/futures/future_fwd.hpp>#include <hpx/runtime/components/make_client.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <cstddef>#include <string>#include <utility>#include <vector>

file **binpacking_distribution_policy.hpp**

#include <hpx/config.hpp>#include <hpx/assert.hpp>#include <hpx/async_distributed/dataflow.hpp>#include <hpx/futures/future.hpp>#include <hpx/performance_counters/performance_counter.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/modules/errors.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/serialization/string.hpp>#include <hpx/serialization/vector.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <cstddef>#include <type_traits>#include <hpx/functional/bind_back.hpp>#include <hpx/pack_traversal/unwrap.hpp>#include <algorithm>#include <cstdint>#include <iterator>#include <string>#include <utility>#include <vector>

file **colocating_distribution_policy.hpp**

#include <hpx/config.hpp>#include <hpx/actions_base/traits/extract_action.hpp>#include <hpx/async_base/launch_policy.hpp>#include <hpx/async_distributed/applier/detail/apply_colocated_callback_fwd.hpp>#inclu <hpx/async_distributed/applier/detail/apply_colocated_fwd.hpp>#include <hpx/async_distributed/applier/detail/apply_impleme <hpx/async_distributed/detail/async_colocated.hpp>#include <hpx/async_distributed/detail/async_colocated_callback.hpp>#in <hpx/async_distributed/detail/async_implementations.hpp>#include <hpx/futures/future.hpp>#include <hpx/futures/traits/promise_local_result.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <algorithm>#include <cstddef>#include <type_traits>#include <utility>#include <vector>

file **component_factory.hpp**

Defines

HPX_REGISTER_COMPONENT(*type*, *name*, *mode*)

Define a component factory for a component type.

This macro is used create and to register a minimal component factory for a component type which allows it to be remotely created using the hpx::new_<> function.

This macro can be invoked with one, two or three arguments

Parameters

- `type`: The *type* parameter is a (fully decorated) type of the component type for which a factory should be defined.
- `name`: The *name* parameter specifies the name to use to register the factory. This should uniquely (system-wide) identify the component type. The *name* parameter must conform to the C++ identifier rules (without any namespace). If this parameter is not given, the first parameter is used.
- `mode`: The *mode* parameter has to be one of the defined enumeration values of the enumeration *hpx::components::factory_state_enum*. The default for this parameter is *hpx::components::factory_enabled*.

file **copy_component.hpp**

#include <hpx/config.hpp>#include <hpx/actions_base/plain_action.hpp>#include

<hpx/async_distributed/async.hpp>#include <hpx/async_distributed/detail/async_colocated.hpp>#include <hpx/components_base/traits/is_component.hpp>#include <hpx/futures/future.hpp>#include <hpx/runtime/components/server/copy_component.hpp>#include <hpx/runtime/components/stubs/runtime_support.hpp>#includ <hpx/modules/errors.hpp>#include <hpx/async_distributed/detail/async_colocated_fwd.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/runtime/components/server/runtime_support.hpp>#include <hpx/actions/base_action.hpp>#include <hpx/actions/transfer_action.hpp>#include <hpx/actions/transfer_continuation_action.hpp>#include <hpx/actions_base/component_action.hpp>#include <hpx/assert.hpp>#include <hpx/modules/plugin.hpp>#include <hpx/modules/program_options.hpp>#include <hpx/performance_counters/counters.hpp>#include <hpx/plugins/plugin_factory_base.hpp>#include <hpx/runtime_configuration/plugin_registry_base.hpp>#include <hpx/runtime_configuration/ini.hpp>#include <hpx/type_support/pack.hpp>#include <hpx/runtime/components/server/create_component.hpp>#include <hpx/runtime/components/server/create_component_fwd.hpp>#include <hpx/runtime/naming/address.hpp>#include <cstddef>#include <cstdint>#include <utility>#include <vector>#include <hpx/runtime/components/server/component_heap.hpp>#include <hpx/static_reinit/reinitializable_static.hpp>#include <sstream>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/parcelset/locality.hpp>#include <hpx/runtime/parcelset_fwd.hpp>#include <hpx/serialization/map.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/iterator_support/traits/is_iterator.hpp>#include <map>#include <memory>#include <string>#include <type_traits>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/runtime_configuration/static_factory_data.hpp>#include <hpx/synchronization/condition_variable.hpp>#include <hpx/synchronization/mutex.hpp>#include <hpx/synchronization/spinlock.hpp>#include <hpx/traits/action_does_termination_detection.hpp>#include <atomic>#include <condition_variable>#include <list>#include <mutex>#include <set>#include <thread>#include <hpx/runtime/naming/name.hpp>#include <hpx/serialization/vector.hpp>#include <hpx/type_support/decay.hpp>#include <hpx/runtime/get_ptr.hpp>#include <hpx/async_base/launch_policy.hpp>#include <hpx/components_base/get_lva.hpp>#include <hpx/components_base/traits/component_pin_support.hpp>#include <hpx/functional/bind_back.hpp>#include <hpx/runtime/agas/gva.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime_fwd.hpp>#include <hpx/traits/component_type_is_compatible.hpp>#include <hpx/futures/traits/get_remote_result.hpp>

file **default_distribution_policy.hpp**

#include <hpx/config.hpp>#include <hpx/actions_base/actions_base_support.hpp>#include <hpx/actions_base/traits/extract_action.hpp>#include <hpx/assert.hpp>#include <hpx/async_base/launch_policy.hpp>#include <hpx/async_distributed/applier/apply.hpp>#include <hpx/async_distributed/dataflow.hpp>#include <hpx/futures/future.hpp>#include <hpx/futures/traits/promise_local_result.hpp>#include <hpx/lcos/packaged_action.hpp>#include <hpx/actions_base/traits/action_priority.hpp>#include <hpx/allocator_support/internal_allocator.hpp>#include <hpx/async_distributed/applier/apply_callback.hpp>#include <hpx/lcos/promise.hpp>#include <hpx/memory/intrusive_ptr.hpp>#include <hpx/modules/errors.hpp>#include <hpx/modules/memory.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/traits/action_was_object_migrated.hpp>#include <hpx/components_base/pinned_ptr.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/type_support/detail/wrap_int.hpp>#include <type_traits>#include <utility>#include <hpx/traits/component_supports_migration.hpp>#include <hpx/traits/component_type_is_compatible.hpp>#include <boost/asio/error.hpp>#include <exception>#include <memory>#include <hpx/modules/execution.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/serialization/shared_ptr.hpp>#include <hpx/serialization/vector.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <algorithm>#include <cstddef>#include <vector>

file **migrate_component.hpp**

#include <hpx/config.hpp>#include <hpx/actions_base/plain_action.hpp>#include <hpx/async_distributed/async.hpp>#include <hpx/async_distributed/detail/async_colocated.hpp>#include <hpx/components_base/traits/is_component.hpp>#include <hpx/futures/future.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/server/migrate_component.hpp>#include

<hpx/runtime/agas/interface.hpp>#include <hpx/runtime/components/stubs/runtime_support.hpp>#include <hpx/runtime/get_ptr.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/traits/component_supports_migration.hpp>#include <cstdint>#include <memory>#include <utility>#include <hpx/runtime/components/target_distribution_policy.hpp>#include <hpx/actions_base/actions_base_support.hpp>#include <hpx/actions_base/traits/extract_action.hpp>#include <hpx/async_base/launch_policy.hpp>#include <hpx/async_distributed/applier/detail/apply_implementations_fwd.hpp>#include <hpx/async_distributed/dataflow.hpp>#include <hpx/async_distributed/detail/async_implementations_fwd.hpp>#include <hpx/futures/traits/promise_local_result.hpp>#include <hpx/lcos/packaged_action.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <algorithm>#include <cstddef>#include <type_traits>#include <vector>

file **new.hpp**

#include <hpx/config.hpp>#include <hpx/async_base/launch_policy.hpp>#include <hpx/components_base/traits/is_component.hpp>#include <hpx/futures/future.hpp>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/components/default_distribution_policy.hpp>#include <hpx/actions_base/actions_base_support.hpp>#include <hpx/actions_base/traits/extract_action.hpp>#include <hpx/assert.hpp>#include <hpx/async_distributed/applier/apply.hpp>#include <hpx/async_distributed/dataflow.hpp>#include <hpx/futures/traits/promise_local_result.hpp>#include <hpx/lcos/packaged_action.hpp>#include <hpx/modules/execution.hpp>#include <hpx/runtime/components/stubs/stub_base.hpp>#include <hpx/runtime/find_here.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/serialization/shared_ptr.hpp>#include <hpx/serialization/vector.hpp>#include <hpx/traits/is_distribution_policy.hpp>#include <algorithm>#include <cstddef>#include <memory>#include <type_traits>#include <utility>#include <vector>#include <hpx/runtime/components/server/create_component.hpp>#include <hpx/traits/is_client.hpp>#include <hpx/type_support/lazy_enable_if.hpp>

file **find_here.hpp**

#include <hpx/config.hpp>#include <hpx/modules/errors.hpp>#include <hpx/runtime/naming/id_type.hpp>

file **find_localities.hpp**

#include <hpx/config.hpp>#include <hpx/modules/errors.hpp>#include <hpx/runtime/components/component_type.hpp>#includ <hpx/runtime/naming/id_type.hpp>#include <vector>

file **get_colocation_id.hpp**

#include <hpx/async_base/launch_policy.hpp>#include <hpx/futures/future_fwd.hpp>#include <hpx/modules/errors.hpp>#include <hpx/runtime/naming/id_type.hpp>

file **get_locality_name.hpp**

#include <hpx/config.hpp>#include <hpx/modules/futures.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <string>

file **get_ptr.hpp**

#include <hpx/config.hpp>#include <hpx/assert.hpp>#include <hpx/async_base/launch_policy.hpp>#include <hpx/components_base/get_lva.hpp>#include <hpx/components_base/traits/component_pin_support.hpp>#include <hpx/functional/bind_back.hpp>#include <hpx/modules/errors.hpp>#include <hpx/runtime/agas/gva.hpp>#include <hpx/runtime/components/component_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/util/ios_flags_saver.hpp>#include <cstdint>#include <hpx/runtime/components/client_base.hpp>#include <hpx/runtime/naming/address.hpp>#include <hpx/runtime/naming_fwd.hpp>#include <hpx/serialization/serialization_fwd.hpp>#include <hpx/serialization/traits/is_bitwise_serializable.hpp>#include <iosfwd>#include <hpx/config/warnings_prefix.hpp>#include <hpx/config/warnings_suffix.hpp>#include <hpx/runtime_fwd.hpp>#include <hpx/runtime_local/runtime_local_fwd.hpp>#include <hpx/traits/component_type_is_compatible.hpp>#include <memory>

file **unmanaged.hpp**
#include <hpx/config.hpp>#include <hpx/runtime/naming/name.hpp>

file **set_parcel_write_handler.hpp**
#include <hpx/config.hpp>

file **trigger_lco.hpp**
#include <hpx/config.hpp>#include <hpx/actions/actions_fwd.hpp>#include <hpx/actions_base/action_priority.hpp>#include <hpx/actions_base/continuation_fwd.hpp>#include <hpx/assert.hpp>#include <hpx/async_distributed/applier/detail/apply_implementations_fwd.hpp>#include <hpx/lcos_fwd.hpp>#include <hpx/components_base/traits/is_component.hpp>#include <hpx/futures/future_fwd.hpp>#include <hpx/futures/traits/promise_local_result.hpp>#include <hpx/futures/traits/promise_remote_result.hpp>#include <vector>#include <hpx/runtime/naming/address.hpp>#include <hpx/runtime/naming/id_type.hpp>#include <hpx/runtime/naming/name.hpp>#include <hpx/type_support/decay.hpp>#include <exception>#include <type_traits>#include <utility>

file **runtime_fwd.hpp**
#include <hpx/config.hpp>#include <hpx/runtime_local/runtime_local_fwd.hpp>

dir **/hpx/source/components/component_storage**

dir **/hpx/source/components/component_storage/include/hpx/components/component_storage**

dir **/hpx/source/hpx/runtime/components**

dir **/hpx/source/components/component_storage/include/hpx/components**

dir **/hpx/source/components**

dir **/hpx/source/hpx**

dir **/hpx/source/components/component_storage/include/hpx**

dir **/hpx/source/components/component_storage/include**

dir **/hpx/source/hpx/runtime/naming**

dir **/hpx/source/hpx/runtime**

dir **/hpx/source**

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.

Header hpx/actions/action_support.hpp

Header hpx/actions/actions_fwd.hpp

Header hpx/actions/base_action.hpp

Header hpx/actions/register_action.hpp

Header hpx/actions/transfer_action.hpp

Header hpx/actions/transfer_base_action.hpp

Header hpx/actions/transfer_continuation_action.hpp

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.

Header hpx/actions_base/action_priority.hpp

namespace hpx

namespace actions

Functions

template<typename **Action**>
threads::*thread_priority* **action_priority**()

Header hpx/actions_base/actions_base_fwd.hpp

Header hpx/actions_base/actions_base_support.hpp

Header hpx/actions_base/basic_action.hpp

Defines

HPX_REGISTER_ACTION_DECLARATION(...)
Declare the necessary component action boilerplate code.

The macro *HPX_REGISTER_ACTION_DECLARATION* can be used to declare all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to declare the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

```
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server
```

(continues on next page)

(continued from previous page)

```
        : public hpx::components::simple_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);
```

Example:

Note This macro has to be used once for each of the component actions defined using one of the *HPX_DEFINE_COMPONENT_ACTION* macros. It has to be visible in all translation units using the action, thus it is recommended to place it into the header file defining the component.

HPX_REGISTER_ACTION_DECLARATION_(...)

HPX_REGISTER_ACTION_DECLARATION_1(*action*)

HPX_REGISTER_ACTION(...)

Define the necessary component action boilerplate code.

The macro *HPX_REGISTER_ACTION* can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

This macro can be invoked with an optional second parameter. This parameter specifies a unique name of the action to be used for serialization purposes. The second parameter has to be specified if the first parameter is not usable as a plain (non-qualified) C++ identifier, i.e. the first parameter contains special characters which cannot be part of a C++ identifier, such as ‘<’, ‘>’, or ‘:’.

Note This macro has to be used once for each of the component actions defined using one of the *HPX_DEFINE_COMPONENT_ACTION* or *HPX_DEFINE_PLAIN_ACTION* macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note Only one of the forms of this macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID* should be used for a particular action, never both.

HPX_REGISTER_ACTION_ID(*action*, *actionname*, *actionid*)

Define the necessary component action boilerplate code and assign a predefined unique id to the action.

The macro *HPX_REGISTER_ACTION* can be used to define all the boilerplate code which is required for proper functioning of component actions in the context of HPX.

The parameter *action* is the type of the action to define the boilerplate for.

The parameter *actionname* specifies an unique name of the action to be used for serialization purposes. The second parameter has to be usable as a plain (non-qualified) C++ identifier, it should not contain special characters which cannot be part of a C++ identifier, such as '<', '>', or ':'.

The parameter *actionid* specifies an unique integer value which will be used to represent the action during serialization.

Note This macro has to be used once for each of the component actions defined using one of the *HPX_DEFINE_COMPONENT_ACTION* or global actions *HPX_DEFINE_PLAIN_ACTION* macros. It has to occur exactly once for each of the actions, thus it is recommended to place it into the source file defining the component.

Note Only one of the forms of this macro *HPX_REGISTER_ACTION* or *HPX_REGISTER_ACTION_ID* should be used for a particular action, never both.

Header `hpx/actions_base/basic_action_fwd.hpp`

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

Header `hpx/actions_base/component_action.hpp`

Defines

HPX_DEFINE_COMPONENT_ACTION(...)

Registers a member function of a component as an action type with HPX.

The macro *HPX_DEFINE_COMPONENT_ACTION* can be used to register a member function of a component as an action type named *action_type*.

The parameter *component* is the type of the component exposing the member function *func* which should be associated with the newly defined action type. The parameter `action_type` is the name of the action type to register with HPX.

```
namespace app
{
    // Define a simple component exposing one action 'print_greeting'
    class HPX_COMPONENT_EXPORT server
      : public hpx::components::simple_component_base<server>
    {
        void print_greeting() const
        {
            hpx::cout << "Hey, how are you?\n" << hpx::flush;
        }
```

(continues on next page)

(continued from previous page)

```
        // Component actions need to be declared, this also defines the
        // type 'print_greeting_action' representing the action.
        HPX_DEFINE_COMPONENT_ACTION(server, print_greeting,
            print_greeting_action);
    };
}
```

Example:

The first argument must provide the type name of the component the action is defined for.

The second argument must provide the member function name the action should wrap.

The default value for the third argument (the typename of the defined action) is derived from the name of the function (as passed as the second argument) by appending '_action'. The third argument can be omitted only if the second argument with an appended suffix '_action' resolves to a valid, unqualified C++ type name.

Note The macro *HPX_DEFINE_COMPONENT_ACTION* can be used with 2 or 3 arguments. The third argument is optional.

Header `hpx/actions_base/continuation_fwd.hpp`

namespace hpx

namespace actions

Functions

template<typename **Result**, typename **RemoteResult**, typename **F**, typename ...**Ts**>
void **trigger** (typed_continuation<*Result*, *RemoteResult*> &&*cont*, *F* &&*f*, *Ts*&&... *vs*)

Header `hpx/actions_base/plain_action.hpp`

Defines

HPX_DEFINE_PLAIN_ACTION (...)
Defines a plain action type.

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }

    // This will define the action type 'app::some_global_action' which
    // represents the function 'app::some_global_function'.
    HPX_DEFINE_PLAIN_ACTION(some_global_function, some_global_action);
}
```

Example:

Note Usually this macro will not be used in user code unless the intent is to avoid defining the action_type in global namespace. Normally, the use of the macro *HPX_PLAIN_ACTION* is recommended.

Note The macro *HPX_DEFINE_PLAIN_ACTION* can be used with 1 or 2 arguments. The second argument is optional. The default value for the second argument (the typename of the defined action) is derived from the name of the function (as passed as the first argument) by appending '_action'. The second argument can be omitted only if the first argument with an appended suffix '_action' resolves to a valid, unqualified C++ type name.

HPX_DECLARE_PLAIN_ACTION(...)
Declares a plain action type.

HPX_PLAIN_ACTION(...)
Defines a plain action type based on the given function *func* and registers it with HPX.

The macro *HPX_PLAIN_ACTION* can be used to define a plain action (e.g. an action encapsulating a global or free function) based on the given function *func*. It defines the action type *name* representing the given function. This macro additionally registers the newly define action type with HPX.

The parameter `func` is a global or free (non-member) function which should be encapsulated into a plain action. The parameter `name` is the name of the action type defined by this macro.

```
namespace app
{
    void some_global_function(double d)
    {
        cout << d;
    }
}

// This will define the action type 'some_global_action' which 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*

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.

Header `hpx/actions_base/preassigned_action_id.hpp`

Header `hpx/actions_base/traits/action_continuation.hpp`

namespace hpx

namespace traits

template<typename **Action**, typename **Enable** = void>
struct action_continuation
#include <action_continuation.hpp>

Public Types

typedef *hpx*::*traits*::*extract_action*<Action>::type::continuation_type **type**

Header `hpx/actions_base/traits/action_priority.hpp`

namespace hpx

namespace traits

template<typename **Action**, typename **Enable** = void>
struct action_priority
#include <action_priority.hpp>

Public Static Attributes

constexpr *threads*::*thread_priority* **value** = *threads*::*thread_priority_default*

Header `hpx/actions_base/traits/action_remote_result.hpp`

Header `hpx/actions_base/traits/action_select_direct_execution.hpp`

namespace hpx

namespace traits

template<typename **Action**, typename **Enable** = void>
struct action_select_direct_execution
#include <action_select_direct_execution.hpp>

Public Static Functions

static constexpr *launch* **call**(*launch policy*, *naming*::*address_type lva*)

Header `hpx/actions_base/traits/action_stacksize.hpp`

namespace hpx

namespace traits

template<typename **Action**, typename **Enable** = void>
struct action_stacksize
#include <action_stacksize.hpp>

Public Static Attributes

constexpr *threads::thread_stacksize* **value** = *threads::thread_stacksize_default*

Header `hpx/actions_base/traits/extract_action.hpp`

namespace hpx

namespace traits

template<typename **Action**, typename **Enable** = void>
struct extract_action
#include <extract_action.hpp>

Public Types

template<>
using type = **typename** Action::derived_type

template<>
using result_type = **typename** *type*::result_type

template<>
using local_result_type = **typename** *type*::local_result_type

template<>
using remote_result_type = **typename** *type*::remote_result_type

affinity

The contents of this module can be included with the header `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 `hpx/modules/affinity.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.

Header `hpx/affinity/affinity_data.hpp`

Header `hpx/affinity/parse_affinity_options.hpp`

namespace hpx

namespace threads

Functions

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.

Header `hpx/algorithms/traits/is_value_proxy.hpp`

Header `hpx/algorithms/traits/projected.hpp`

template<typename **Iterator**>
struct projected_iterator<*Iterator*, **typename** *std*::enable_if<is_segmented_iterator<*Iterator*>::value>::type>
#include <projected.hpp>

Public Types

typedef segmented_iterator_traits<Iterator>::local_iterator **local_iterator**

typedef segmented_local_iterator_traits<*local_iterator*>::local_raw_iterator **type**

template<typename **Iterator**>
struct projected_iterator<*Iterator*, **typename** *hpx*::*util*::always_void<**typename** *hpx*::*util*::decay<*Iterator*>::type::proxy
#include <projected.hpp>

Public Types

typedef *hpx*::*util*::decay<Iterator>::type::proxy_type **type**

namespace hpx

namespace parallel

namespace traits

template<typename **Proj**, typename **Iter**>
struct projected
#include <projected.hpp>

Public Types

typedef *hpx*::*util*::decay<Proj>::type **projector_type**

typedef *hpx*::*traits*::projected_iterator<Iter>::type **iterator_type**

namespace traits

template<typename **T**, typename **Enable** = void>
struct projected_iterator
#include <projected.hpp>

Public Types

typedef *hpx*::*util*::decay<T>::type **type**

template<typename **Iterator**>
struct projected_iterator<*Iterator*, **typename** *hpx*::*util*::always_void<**typename** *hpx*::*util*::decay<*Iterator*>:
#include <projected.hpp>

Public Types

typedef *hpx*::*util*::decay<Iterator>::type::proxy_type **type**

template<typename **Iterator**>
struct projected_iterator<*Iterator*, **typename** *std*::enable_if<is_segmented_iterator<*Iterator*>::value>::type>
#include <projected.hpp>

Public Types

typedef segmented_iterator_traits<Iterator>::local_iterator **local_iterator**

typedef segmented_local_iterator_traits<*local_iterator*>::local_raw_iterator **type**

Header `hpx/algorithms/traits/projected_range.hpp`

template<typename **Proj**, typename **Rng**>
struct projected_range<*Proj*, *Rng*, **typename** *std*::enable_if<*hpx*::*traits*::is_range<*Rng*>::value>::type>
#include <projected_range.hpp>

Public Types

typedef *hpx*::*util*::decay<Proj>::type **projector_type**

typedef *hpx*::*traits*::range_iterator<Rng>::type **iterator_type**

namespace hpx

namespace parallel

namespace traits

template<typename **Proj**, typename **Rng**>
struct projected_range<*Proj*, *Rng*, **typename** *std*::enable_if<*hpx*::*traits*::is_range<*Rng*>::value>::type>
#include <projected_range.hpp>

Public Types

typedef *hpx*::*util*::decay<Proj>::type **projector_type**

typedef *hpx*::*traits*::range_iterator<Rng>::type **iterator_type**

Header `hpx/algorithms/traits/segmented_iterator_traits.hpp`

namespace hpx

namespace traits

template<typename **Iterator**, typename **Enable** = void>
struct segmented_iterator_traits
#include <segmented_iterator_traits.hpp>

Public Types

typedef *std*::false_type **is_segmented_iterator**

template<typename **Iterator**, typename **Enable** = void>
struct segmented_local_iterator_traits
#include <segmented_iterator_traits.hpp>

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*)

Header `hpx/parallel/algorithm.hpp`

Header `hpx/parallel/algorithms/adjacent_difference.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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*.

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.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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.
- `Op`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *adjacent_difference* requires *Op* to meet the requirements of *CopyConstructible*.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of the range the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of the range the algorithm will be applied to.
- `dest`: Refers to the beginning of the sequence of elements the results will be assigned to.
- `op`: The binary operator which returns the difference of elements. The signature should be equivalent to the following:

```
bool op(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* must be such that objects of type *FwdIter1* can be dereferenced and then implicitly converted to the dereferenced type of *dest*.

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.

Header hpx/parallel/algorithms/adjacent_find.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = detail::equal_to>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

Searches the range [first, last) for two consecutive identical elements. This version uses the given binary predicate op

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.
- `op`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

```
bool pred(const Type1 &a, const 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 *op*.

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.

Header `hpx/parallel/algorithms/all_any_none.hpp`

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:

```
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.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**, typename **Proj** = *util*::projection_identity>
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 an forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *any_of* requires *F* to meet the requirements of *CopyConstructible*.
- `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:

```
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).

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:

```
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.

Header `hpx/parallel/algorithms/copy.hpp`

`namespace hpx`

Functions

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 *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.

template<typename **`ExPolicy`**, typename **`FwdIter1`**, typename **`Size`**, typename **`FwdIter2`**>
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 *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 *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.

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 [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 *last* - *first* assignments, exactly *last* - *first* applications of the predicate *f*.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of an forward iterator.

- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type.

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.

Header `hpx/parallel/algorithms/count.hpp`

`namespace hpx`

Functions

template<typename **ExPolicy**, typename **FwdIterB**, typename **FwdIterE**, typename **T**, typename **Proj** = *util*::projection_
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.

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:

  ```
  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.

Header `hpx/parallel/algorithms/destroy.hpp`

namespace hpx

Functions

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.

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.

Header `hpx/parallel/algorithms/equal.hpp`

`namespace hpx`

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:

```
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.

Note The two ranges are considered equal if, for every iterator i in the range [first1,last1), *i equals *(first2 + (i - first1)). This overload of equal uses operator== to determine if two elements are equal.

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.

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*, *Pred &&op = Pred*())

Returns true if the range [first1, last1) is equal to the range starting at first2, and false otherwise.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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 an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `op`: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

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

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

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), *i 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.

Header `hpx/parallel/algorithms/exclusive_scan.hpp`

`namespace hpx`

`namespace parallel`

`namespace v1`

Functions

template<typename **`ExPolicy`**, typename **`FwdIter1`**, typename **`FwdIter2`**, typename **`T`**, typename **`Op`**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **`typename`** *util*::detail::algorithm_result<*ExPolicy*, *F*

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *first, . . . , *(first + (i - result) - 1)).

The reduce operations in the parallel *exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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:

```
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<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(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 **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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.
- `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 ith input element in the ith sum.

Return The *exclusive_scan* algorithm returns a *hpx::future<FwdIter2>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter2* otherwise. The *exclusive_scan* algorithm returns the output iterator to the element in the destination range, one past the last element copied.

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.

Header `hpx/parallel/algorithms/fill.hpp`

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 [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.

- `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 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*.

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 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 *count* assignments, for count > 0.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of 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 *first* the algorithm will be applied to.
- `value`: The value to be assigned.

The comparisons in the parallel *fill_n* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

Return The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

Header `hpx/parallel/algorithms/find.hpp`

`namespace hpx`

Functions

template<typename **`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 [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most last - first applications of the operator==().

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `T`: The type of the value to find (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `val`: the value to compare the elements to

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:

```
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* algorithm returns the first element in the range [first, last) that does **not** satisfy the predicate *f*. If no such element exists that does not satisfy the predicate f, the algorithm returns *last*.

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

Returns the last subsequence of elements [first2, last2) found in the range [first, last) using the given predicate *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* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `Proj`: The type of an optional projection function. This defaults to *util::projection_identity* and is applied to the elements of type dereferenced *FwdIter1* and dereferenced *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 *true* if the elements should be treated as equal. The signature should be equivalent to the following:

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

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

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* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter1`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an forward iterator.

- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *equal* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `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:

```
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*.

Header `hpx/parallel/algorithms/for_each.hpp`

`namespace hpx`

Functions

template<typename **`InIter`**, typename **`F`**>
F **`for_each`** (*InIter first*, *InIter last*, *F &&f*)

Applies *f* to the result of dereferencing every iterator in the range [first, last).

If *f* returns a result, the result is ignored.

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 *MoveConstructible*.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

The signature does not need to have const&. The type *Type* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to Type.

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.
- `FwdIte`: 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:

```
<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* 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 *MoveConstructible*.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at *first* the algorithm will be applied to.

- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

  ```
  <ignored> pred(const Type &a);
  ```

  The signature does not need to have const&. The type *Type* must be such that an object of type *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). 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.

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.

Header `hpx/parallel/algorithms/for_loop.hpp`

`namespace hpx`

Functions

template<typename **I**, typename ...**Args**>
void **for_loop**(**typename** *std*::decay<*I*>::type *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 *parallel::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:

  ```
  <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.

template<typename **ExPolicy**, typename **I**, typename ...**Args**>
util::detail::algorithm_result<*ExPolicy*>::type **for_loop**(*ExPolicy* &&*policy*, **typename** *std*::decay<*I*>::type *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:

```
<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* 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 **I**, typename **S**, typename ...**Args**>
void **for_loop_strided**(**typename** *std*::decay<*I*>::type *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 *parallel::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.
- `S`: The type of the stride variable. This should be an integral type.
- `Args`: A parameter pack, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `stride`: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- `args`: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

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

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

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 **S**, typename ...**Args**>
util::detail::algorithm_result<*ExPolicy*>::type **for_loop_strided**(*ExPolicy* &&*policy*, **typename** *std*::decay<*I*>::type *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, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `stride`: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- `args`: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects.

The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

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

The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

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 *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **I**, typename **Size**, typename ...**Args**>
void **for_loop_n** (*I first*, *Size size*, *Args&&... args*)

The for_loop 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 *parallel::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, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.

- `size`: Refers to the number of items the algorithm will be applied to.
- `args`: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

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

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

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 **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:

```
<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 *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

template<typename **I**, typename **Size**, typename **S**, typename ...**Args**>
void **for_loop_n_strided**(*I first*, *Size size*, *S stride*, *Args&&... args*)

The for_loop_n_strided implements loop functionality over a range specified by integral or iterator bounds. For the iterator case, these algorithms resemble for_each from the Parallelism TS, but leave to the programmer when and if to dereference the iterator.

The execution of for_loop without specifying an execution policy is equivalent to specifying *parallel::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, it's last element is a function object to be invoked for each iteration, the others have to be either conforming to the induction or reduction concept.

Parameters

- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `size`: Refers to the number of items the algorithm will be applied to.
- `stride`: Refers to the stride of the iteration steps. This shall have non-zero value and shall be negative only if I has integral type or meets the requirements of a bidirectional iterator.
- `args`: The last element of this parameter pack is the function (object) to invoke, while the remaining elements of the parameter pack are instances of either induction or reduction objects. The function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last) should expose a signature equivalent to:

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

The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

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 **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 [first, last) should expose a signature equivalent to:

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

  The signature does not need to have const&. It will receive the current value of the iteration variable and one argument for each of the induction or reduction objects passed to the algorithms, representing their current values.

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_strided* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise.

Header `hpx/parallel/algorithms/for_loop_induction.hpp`

namespace hpx

namespace parallel

namespace v2

Functions

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 *value* of that type and, optionally, a stride.

For each element in the input range, a looping algorithm over input sequence *S* computes an induction value from an induction variable and ordinal position *p* within *S* by the formula i + p * stride if a stride was specified or i + p otherwise. This induction value is passed to the element access function.

If the *value* argument to *induction* is a non-const lvalue, then that lvalue becomes the live-out object for the returned induction object. For each induction object that has a live-out object, the looping algorithm assigns the value of i + n * stride to the live-out object upon return, where *n* is the number of elements in the input range.

Return This returns an induction object with value type *T*, initial value *value*, and (if specified) stride *stride*. If *T* is an lvalue of non-const type, *value* is used as the live-out object for the induction object; otherwise there is no live-out object.

Template Parameters
- `T`: The value type to be used by the induction object.

Parameters
- `value`: [in] The initial value to use for the induction object
- `stride`: [in] The (optional) stride to use for the induction object (default: 1)

Header `hpx/parallel/algorithms/for_loop_reduction.hpp`

namespace hpx

namespace parallel

namespace v2

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.

Header `hpx/parallel/algorithms/generate.hpp`

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 *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*.

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 *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*.

Header `hpx/parallel/algorithms/includes.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::less>

std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, b

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:

```
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 *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.

Header `hpx/parallel/algorithms/inclusive_scan.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**, typename **T**>
util::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **inclusive_scan**(*ExPolicy &&policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *Op &&op*, *T init*)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(op, init, *first, . . . , *(first + (i - result))).

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.
- `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:

```
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.

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:

```
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.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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 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 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(+, a1, . . . , aK)
  - GENERALIZED_NONCOMMUTATIVE_SUM(+, aM, . . . , aN) where 1 < K+1 = M <= N.

Header `hpx/parallel/algorithms/is_heap.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **RandIter**, typename **Comp** = detail::less, typename **Proj** = *util*::*project*
util::detail::algorithm_result<*ExPolicy*, bool>::type **is_heap** (*ExPolicy &&policy*, *RandIter first*, *RandIter 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.
- `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).
- `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 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.

template<typename **ExPolicy**, typename **RandIter**, typename **Comp** = detail::less, typename **Proj** = *util*::*project*
util::detail::algorithm_result<*ExPolicy*, *RandIter*>::type **is_heap_until**(*ExPolicy* &&*policy*, *RandIter first*, *RandIter last*, *Comp* &&*comp* = *Comp*(), *Proj* &&*proj* = *Proj*())

Returns the upper bound of the largest range beginning at *first* which is a max heap. That is, the last iterator *it* for which range [first, it) is a max heap. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

Note Complexity: Performs at most N applications of the comparison *comp*, at most 2 * N applications of the projection *proj*, where N = last - first.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (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).
- `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 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.

Header hpx/parallel/algorithms/is_partitioned.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, b

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* = distance(first, last).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- `pred`: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

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.

Header `hpx/parallel/algorithms/is_sorted.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = detail::less>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, b

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* = distance(first, last). *S* = number of partitions

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used for the This iterator type must meet the requirements of a forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *is_sorted* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::less<>

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements of that the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements of that the algorithm will be applied to.
- `pred`: Refers to the binary predicate which returns true if the first argument should be treated as less than the second argument. The signature of the function should be equivalent to

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to Type.

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.

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = detail::less>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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

```
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.

Header `hpx/parallel/algorithms/lexicographical_compare.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::less>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, b

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 *CopyConstructible*. This defaults to std::less<>

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `pred`: Refers to the comparison function that the first and second ranges will be applied to

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.

Header `hpx/parallel/algorithms/merge.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **RandIter1**, typename **RandIter2**, typename **RandIter3**, typename *util*::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in1 (*RandIter1*), *tag*::in2 *RandIter2*, *tag*::out*RandIter3*>>::type **merge***ExPolicy* &&*policy*, *RandIter1 first1*, *RandIter1 last1*, *RandIter2 first2*, *RandIter2 last2*, *RandIter3 dest*, *Comp* &&*comp* = *Comp*(), *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*()Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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<>
- `Proj1`: The type of an optional projection function to be used for elements of the first range. This defaults to 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 first range of elements the algorithm will be applied to.
- `last1`: Refers to the end of the first range of elements the algorithm will be applied to.
- `first2`: Refers to the beginning of the second range of elements the algorithm will be applied to.
- `last2`: Refers to the end of the second range of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `comp`: *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

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

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter1* and *RandIter2* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- `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<tagged_tuple<tag::in1(RandIter1), tag::in2(RandIter2), tag::out(RandIter3)> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_tuple<tag::in1(RandIter1), tag::in2(RandIter2), tag::out(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.

template<typename **ExPolicy**, typename **RandIter**, typename **Comp** = detail::less, typename **Proj** = *util*::*project*

util::detail::algorithm_result<*ExPolicy*, *RandIter*>::type **inplace_merge**(*ExPolicy* &&*policy*, *RandIter first*, *RandIter middle*, *RandIter 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.
- `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.
- `first`: Refers to the beginning of the first sorted range the algorithm will be applied to.
- `middle`: Refers to the end of the first sorted range and the beginning of the second sorted range the algorithm will be applied to.
- `last`: Refers to the end of the second sorted range the algorithm will be applied to.
- `comp`: *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

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

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *RandIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- `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*

Header `hpx/parallel/algorithms/minmax.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **Proj** = *util*::*projection_identity*, typename **F** = detai
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 = 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:

```
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.

template<typename **ExPolicy**, typename **FwdIter**, typename **Proj** = *util*::*projection_identity*, typename **F** = detai
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 = 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:

```
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 *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 *max_element* algorithm returns a *hpx::future<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *max_element* algorithm returns the iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

template<typename **ExPolicy**, typename **FwdIter**, typename **Proj** = *util*::*projection_identity*, typename **F** = detai
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*(floor(3/2*(N-1)), 0) applications of the predicate, where N = std::distance(first, last).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `FwdIter`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *minmax_element* requires *F* to meet the requirements of *CopyConstructible*.
- `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:

```
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.

Header `hpx/parallel/algorithms/mismatch.hpp`

namespace hpx

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>

util::detail::algorithm_result<*ExPolicy*, *std*::pair<*FwdIter1*, *FwdIter2*>>::type **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:

```
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), *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.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to>
util::detail::algorithm_result<*ExPolicy*, *std*::pair<*FwdIter1*, *FwdIter2*>>::type **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 an forward iterator.
- `FwdIter2`: The type of the source iterators used for the second range (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *mismatch* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>

Parameters

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

- `op`: The binary predicate which returns true if the elements should be treated as mismatch. The signature of the predicate function should be equivalent to the following:

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

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

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).

Header `hpx/parallel/algorithms/move.hpp`

`namespace hpx`

Functions

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.

Header `hpx/parallel/algorithms/partition.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **BidirIter**, typename **F**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, *BidirIter*>::type **stable_partition**(*ExPolicy* &&*policy*, *BidirIter first*, *BidirIter last*, *F* &&*f*, *Proj* &&*proj* = *Proj*())

Permutes the elements in the range [first, last) such that there exists an iterator i such that for every iterator j in the range [first, i) INVOKE(f, INVOKE (proj, *j)) != false, and for every iterator k in the range [i, last), INVOKE(f, INVOKE (proj, *k)) == false

The invocations of *f* in the parallel *stable_partition* algorithm invoked with an execution policy object of type *sequenced_policy* executes in sequential order in the calling thread.

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.
- `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 [first, last). The signature of this predicate should be equivalent to:

```
bool fun(const Type &a);
```

The signature does not need to have const&. The type *Type* must be such that an object of type *BidirIter* can be dereferenced and then implicitly converted to *Type*.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

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 [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 **ExPolicy**, typename **FwdIter**, typename **Pred**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **partition**(*ExPolicy* &&*policy*, *FwdIter first*, *FwdIter last*, *Pred* &&*pred*, *Proj* &&*proj* = *Proj*())

Reorders the elements in the range [first, last) in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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:

```
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 **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pre**
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in(*FwdIter1*), *tag*::out1
FwdIter2, *tag*::out2*FwdIter3*>>::type **partition_copy***ExPolicy* &&*policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest_true*, *FwdIter3 dest_false*, *Pred* &&*pred*, *Proj* &&*proj* = *Proj*()Copies the elements in the range, defined by [first, last), to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred*, are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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:

```
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.

Header `hpx/parallel/algorithms/reduce.hpp`

`namespace hpx`

Functions

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.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIter* can be dereferenced and then implicitly converted to any of those types.

- `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 over 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 **FwdIter**, typename **T**>
util::detail::algorithm_result<*ExPolicy*, *T*>::type **reduce**(*ExPolicy &&policy*, *FwdIter first*, *FwdIter last*, *T init*)

Returns GENERALIZED_SUM(+, init, *first, …, *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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 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.

Header `hpx/parallel/algorithms/reduce_by_key.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **RanIter**, typename **RanIter2**, typename **FwdIter1**, typename **FwdI**
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_outp*, *FwdIter2 values_outpu*, *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:

```
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>* otherwise.

Header `hpx/parallel/algorithms/remove.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **remove_if**(*ExPolicy* &&*policy*, *FwdIter first*, *FwdIter last*, *Pred* &&*pred*, *Proj* &&*proj* = *Proj*())

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* 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 (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*.
- `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<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.

template<typename **ExPolicy**, typename **FwdIter**, typename **T**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **remove**(*ExPolicy* &&*policy*, *FwdIter* *first*, *FwdIter* *last*, *T* **const** &*value*, *Proj* &&*proj* = *Proj*())

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==() 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 (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*.
- `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<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.

Header `hpx/parallel/algorithms/remove_copy.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Proj** = *util*::*p*
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<*FwdIter1*, *FwdIter2*>>::type **remove_copy** (*ExPolicy* &&*policy*, *FwdIter1* *first*, *FwdIter1* *last*, *FwdIter2* *dest*, *T* **const** &*val*, *Proj* &&*proj* = *Proj*())

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 val. 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(proj, *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.
- `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`: Refers to the beginning of the destination range.
- `val`: Value to be removed.
- `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_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<tagged_pair<tag::in(FwdIter1), tag::out(FwdIter2)> >* if the execution policy is of type *sequenced_task_policy* or *paral-*

lel_task_policy and returns *tagged_pair<tag::in(FwdIter1), tag::out(FwdIter2)>* otherwise. The *copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **F**, typename **Proj** = *util*::*p*
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<*FwdIter1*, *FwdIter2*>>::type **remove_copy_if** (*ExPolicy* &&*policy*, *FwdIter1* *first*, *FwdIter1* *last*, *FwdIter2* *dest*, *F* &&*f*, *Proj* &&*proj* = *Proj*())

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, INVOKE(proj, *it)) != false.

The assignments in the parallel *remove_copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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.
- `first`: Refers to the beginning of the sequence 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 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.
- `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_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<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 *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.

Header `hpx/parallel/algorithms/replace.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **T1**, typename **T2**, typename **Proj** = *util*::*projection_*
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **replace**(*ExPolicy* &&*policy*, *FwdIter first*, *FwdIter last*, *T1* **const** &*old_value*, *T2* **const** &*new_value*, *Proj* &&*proj* = *Proj*())

Replaces all elements satisfying specific criteria with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first, last) with new_value, when the following corresponding conditions hold: INVOKE(proj, *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.
- `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<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **FwdIter**, typename **F**, typename **T**, typename **Proj** = *util*::*projection_id*
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **replace_if**(*ExPolicy* &&*policy*, *FwdIter first*, *FwdIter last*, *F* &&*f*, *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, 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, INVOKE(proj, *it)) != false

The assignments in the parallel *replace_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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*. (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.
- `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 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<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 **FwdIter1**, typename **FwdIter2**, typename **T1**, typename **T2**, typename
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (*FwdIter1*), *tag*::out
FwdIter2>>::type **replace_copy***ExPolicy* &&*policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *T1* **const** &*old_value*, *T2* **const** &*new_value*, *Proj* &&*proj* = *Proj*()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(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 *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.
- `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.
- `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<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 *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 **FwdIter2**, typename **F**, typename **T**, typename **P**
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (*FwdIter1*), *tag*::out
FwdIter2>>::type **replace_copy_if***ExPolicy* &&*policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *F* &&*f*, *T* **const** &*new_value*, *Proj* &&*proj* = *Proj*()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, 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 *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.
- `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*. (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.
- `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 elements which need to replaced. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.
- `new_value`: Refers to the new value to use as the replacement.
- `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 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 *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.

Header hpx/parallel/algorithms/reverse.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **BidirIter**>
util::detail::algorithm_result<*ExPolicy*, *BidirIter*>::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<BidirIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *BidirIter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **BidirIter**, typename **FwdIter**>
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<*BidirIter*, *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_first`: 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<tagged_pair<tag::in(BidirIter), tag::out(FwdIter)>* > if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(BidirIter), tag::out(FwdIter)>* 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.

Header `hpx/parallel/algorithms/rotate.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**>
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<*FwdIter*, *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<tagged_pair<tag::begin(FwdIter), tag::end(FwdIter)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_pair<tag::begin(FwdIter), tag::end(FwdIter)>* otherwise. The *rotate* algorithm returns the iterator equal to pair(first + (last - new_first), last).

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<*FwdIter1*, *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 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 bidirectional 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 invoked with an execution policy object of 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<tagged_pair<tag::in(FwdIter1), tag::out(FwdIter2)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_pair<tag::in(FwdIter1), tag::out(FwdIter2)>* otherwise. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

Header `hpx/parallel/algorithms/search.hpp`

`namespace hpx`

`namespace parallel`

`namespace v1`

Functions

template<typename **`ExPolicy`**, typename **`FwdIter`**, typename **`FwdIter2`**, typename **`Pred`** = detail::equal_to, type
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **`search`** (*ExPolicy &&policy*, *FwdIter first*, *FwdIter last*, *FwdIter2 s_first*, *FwdIter2 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 input 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<>
- `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

```
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.

template<typename **ExPolicy**, typename **FwdIter**, typename **FwdIter2**, typename **Pred** = detail::equal_to, type
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*(), *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 = distance(s_first, s_last) and N = count.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `FwdIter`: The type of the source iterators used for the first range (deduced). This iterator type must meet the requirements of an input 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

  ```
  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.

Header `hpx/parallel/algorithms/set_difference.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pre**
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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:

```
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.

Header `hpx/parallel/algorithms/set_intersection.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pre**
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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:

  ```
  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.

Header hpx/parallel/algorithms/set_symmetric_difference.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pre**
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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 *CopyConstructible*. This defaults to std::less<>

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first1`: Refers to the beginning of the sequence of elements of the first range the algorithm will be applied to.
- `last1`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `first2`: Refers to the beginning of the sequence of elements of the second range the algorithm will be applied to.
- `last2`: Refers to the end of the sequence of elements of the second range the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `op`: The binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

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

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *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.

Header `hpx/parallel/algorithms/set_union.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **Pre**

std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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:

```
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.

Header `hpx/parallel/algorithms/sort.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **RandomIt**, typename **Compare** = detail::less, typename **Proj** = *util*::*pro*
util::detail::algorithm_result<*ExPolicy*, *RandomIt*>::type **sort** (*ExPolicy* *&&policy*, *RandomIt* *first*, *RandomIt* *last*, *Compare* *&&comp* = *Compare*(), *Proj* *&&proj* = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

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.
- `Iter`: 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<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.

Header `hpx/parallel/algorithms/sort_by_key.hpp`

`namespace hpx`

`namespace parallel`

`namespace v1`

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.

Header `hpx/parallel/algorithms/stable_sort.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **RandomIt**, typename **Sentinel**, typename **Proj** = *util*::*projection_ide*
util::detail::algorithm_result<*ExPolicy*, *RandomIt*>::type **stable_sort** (*ExPolicy* &&*policy*, *RandomIt first*, *Sentinel last*, *Compare* &&*comp* = *Compare*(), *Proj* &&*proj* = *Proj*())

Sorts the elements in the range [first, last) in ascending order. The relative order of equal elements is preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

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.
- `RandomIt`: The type of the source iterators used (deduced). This iterator type must meet the requirements of a random access iterator.
- `Sentinel`: The type of the end iterators used (deduced). This iterator type must meet the requirements of a random access iterator and must be a valid sentinel type 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 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<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.

Header `hpx/parallel/algorithms/swap_ranges.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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*.

Header `hpx/parallel/algorithms/transform.hpp`

`namespace hpx`

`namespace parallel`

`namespace v1`

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **F**, typename **Proj** = *util*::*p*
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (*FwdIter1*) , *tag*::out
FwdIter2>>::type **transform***ExPolicy* &&*policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *F* &&*f*, *Proj* &&*proj* = *Proj*()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*.
- `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`: 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 *FwdIter* 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<tagged_pair<tag::in(FwdIter1), tag::out(FwdIter2)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_pair<tag::in(FwdIter1), tag::out(FwdIter2)>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **F**, ty
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in1 (*FwdIter1*) , *tag*::in2
FwdIter2, *tag*::out*FwdIter3*>>::type **transform***ExPolicy* &&*policy*, *FwdIter1 first1*, *FwdIter1 last1*, *FwdIter2 first2*, *FwdIter3 dest*, *F* &&*f*, *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*()Applies the given function *f* to pairs of elements from two ranges: one defined by [first1,

last1) and the other beginning at first2, and stores the result in another range, beginning at dest.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly *last* - *first* applications of *f*

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the invocations of *f*.
- `FwdIter1`: The type of the source iterators for the first range used (deduced). This iterator type must meet the requirements of 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*.
- `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.
- `dest`: Refers to the beginning of the destination range.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types FwdIter1 and FwdIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *FwdIter3* can be dereferenced and assigned a value of type *Ret*.
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first sequence as a projection operation before the actual predicate *f* is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second sequence as a projection operation before the actual predicate *f* is invoked.

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<tagged_tuple<tag::in1(FwdIter1), tag::in2(FwdIter2), tag::out(FwdIter3)>* > if the execution policy is of type *parallel_task_policy* and returns *tagged_tuple<tag::in1(FwdIter1), tag::in2(FwdIter2), tag::out(FwdIter3)>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **FwdIter3**, typename **F**, ty
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in1 (*FwdIter1*) , *tag*::in2 *FwdIter2*, *tag*::out*FwdIter3*>>::type **transform***ExPolicy* &&*policy*, *FwdIter1 first1*, *FwdIter1 last1*, *FwdIter2 first2*, *FwdIter2 last2*, *FwdIter3 dest*, *F* &&*f*, *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*()Applies the given function *f* to pairs of elements from two ranges: one defined by [first1, last1) and the other beginning at first2, and stores the result in another range, beginning at dest.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly min(last2-first2, last1-first1) 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*.
- `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:

  ```
  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.

Note The algorithm will invoke the binary predicate until it reaches the end of the shorter of the two given input sequences

Return The *transform* algorithm returns a *hpx::future<tagged_tuple<tag::in1(FwdIter1), tag::in2(FwdIter2), tag::out(FwdIter3)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_tuple<tag::in1(FwdIter1), tag::in2(FwdIter2), tag::out(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.

Header `hpx/parallel/algorithms/transform_exclusive_scan.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Op**, typename
util::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **transform_exclusive_scan**(*ExPolicy &&policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *T init*, *Op &&op*, *Conv &&conv*)

Assigns through each iterator *i* in [result, result + (last - first)) the value of GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, conv(*first), …, conv(*(first + (i - result) - 1))).

The reduce operations in the parallel *transform_exclusive_scan* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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 [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.
- `init`: The initial value for the generalized sum.
- `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 *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<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 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.

Header `hpx/parallel/algorithms/transform_inclusive_scan.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Op**, typename **Conv**, typena
util::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **transform_inclusive_scan**(*ExPolicy* &&*policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *Op* &&*op*, *Conv* &&*conv*, *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.
- `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 [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

```
R fun(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

- `init`: The initial value for the generalized sum.
- `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 *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 *exclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the ith input element in the ith sum. If *op* is not mathematically associative, the behavior of *transform_inclusive_scan* may be non-deterministic.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Conv**, typename **Op**>
util::detail::algorithm_result<*ExPolicy*, *FwdIter2*>::type **transform_inclusive_scan**(*ExPolicy* &&*policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *Op* &&*op*, *Conv* &&*conv*)

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 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.
- `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 [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.

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 *exclusive_scan* and *transform_inclusive_scan* is that *transform_inclusive_scan* includes the ith input element in the ith sum.

Header hpx/parallel/algorithms/transform_reduce.hpp

namespace hpx

Functions

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:

```
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:

```
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.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **T**, typename **Reduce**, typename **Con**
util::detail::algorithm_result<*ExPolicy*, *T*>::type **transform_reduce**(*ExPolicy &&policy*, *FwdIter1 first1*, *FwdIter1 last1*, *FwdIter2 first2*, *T init*, *Reduce &&red_op*, *Convert &&conv_op*)

Returns the result of accumulating init with the inner products of the pairs formed by the elements of two ranges starting at first1 and first2.

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 should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.

- `conv_op`: Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to an object for the second argument type of *op1*.

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.

Header `hpx/parallel/algorithms/transform_reduce_binary.hpp`

Header `hpx/parallel/algorithms/uninitialized_copy.hpp`

`namespace hpx`

`namespace parallel`

`namespace v1`

Functions

template<typename **`ExPolicy`**, typename **`FwdIter1`**, typename **`FwdIter2`**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **`typename`** *util*::detail::algorithm_result<*ExPolicy*, *F*

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.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*, *F*

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 *f* to.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at *first* the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

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.

Header `hpx/parallel/algorithms/uninitialized_default_construct.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**>
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 [first, last) 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 *void* otherwise.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>

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.

Header `hpx/parallel/algorithms/uninitialized_fill.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **T**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*>::

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 **ExPolicy**, typename **FwdIter**, typename **Size**, typename **T**>
std::enable_if<*execution*::*is_execution_policy*<*ExPolicy*>::value, **typename** *util*::detail::algorithm_result<*ExPolicy*>::

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<void>*, if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns nothing otherwise.

Header `hpx/parallel/algorithms/uninitialized_move.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**>
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* move 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.

- `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 **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (*FwdIter1*) , *tag*::out *FwdIter2*>>::type **uninitialized_move_n***ExPolicy* &&*policy*, *FwdIter1 first*, *Size count*, *FwdIter2 dest*Moves the elements in the range [first, first + count), starting from first and proceeding to first + count - 1., to another range beginning at dest. If an exception is thrown during the initialization, some objects in [first, first + count) are left in a valid but unspecified state.

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 forward iterator.
- `Size`: The type of the argument specifying the number of elements to apply *f* to.
- `FwdIter2`: The type of the iterator representing the destination range (deduced). This iterator type must meet the requirements of a forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `count`: Refers to the number of elements starting at *first* the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

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<std::pair<FwdIter1, FwdIter2>>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *std::pair<FwdIter1, FwdIter2>* otherwise. The *uninitialized_move_n* algorithm returns the pair of the input iterator to the element past in the source range and an output iterator to the element in the destination range, one past the last element moved.

Header hpx/parallel/algorithms/uninitialized_value_construct.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**>
util::detail::algorithm_result<*ExPolicy*>::type **uninitialized_value_construct** (*ExPolicy &&policy*, *FwdIter first*, *FwdIter last*)

Constructs objects of type typename iterator_traits<ForwardIt>::value_type in the uninitialized storage designated by the range [first, last) by default-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 *void* otherwise.

template<typename **ExPolicy**, typename **FwdIter**, typename **Size**>
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 default-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 *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.

Header `hpx/parallel/algorithms/unique.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **FwdIter**, typename **Pred** = detail::equal_to, typename **Proj** = *util*::*proj*
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **unique**(*ExPolicy* &&*policy*, *FwdIter* *first*, *FwdIter* *last*, *Pred* &&*pred* = *Pred*(), *Proj* &&*proj* = *Proj*())

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range.

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.
- `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 the end of the sequence of elements the algorithm will be applied to.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

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

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter* can be dereferenced and then implicitly converted to both *Type1* and *Type2*
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *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.

template<typename **ExPolicy**, typename **FwdIter1**, typename **FwdIter2**, typename **Pred** = detail::equal_to, typ
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (*FwdIter1*), *tag*::out
FwdIter2>>::type **unique_copy***ExPolicy* &&*policy*, *FwdIter1 first*, *FwdIter1 last*, *FwdIter2 dest*, *Pred* &&*pred* = *Pred*(), *Proj* &&*proj* = *Proj*()Copies the elements from the range [first, last), to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

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

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *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<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 *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

Header `hpx/parallel/container_algorithms.hpp`

Header `hpx/parallel/container_algorithms/all_any_none.hpp`

namespace hpx

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.

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **Proj** = *util*::projection_identity>
util::detail::algorithm_result<*ExPolicy*, bool>::type **any_of**(*ExPolicy &&policy*, *Rng &&rng*, *F &&f*, *Proj &&proj* = *Proj*())

Checks if unary predicate *f* returns true for at least one element in the range *rng*.

The application of function objects in parallel algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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 *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 **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:

```
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.

Header hpx/parallel/container_algorithms/copy.hpp

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.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter**>

hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*ranges*::copy_result<**typename** *hpx*::*traits*::range_traits<*Rng*>:

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.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Size**, typename **FwdIter2**>
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*ranges*::copy_n_result<*FwdIter1*, *FwdIter2*>>::type **copy_n** (*E*
&
ic
F
fi
S
co
F
de

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 *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 *hpx::future<ranges::copy_n_result<FwdIter1, FwdIter2> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *ranges::copy_n_result<FwdIter1, FwdIter2>* otherwise. The *copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **FwdIter1**, typename **Sent1**, typename **FwdIter**, typename **F**, typename **P**
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*ranges*::copy_if_result<**typename** *hpx*::*traits*::range_traits<Rn

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:

  ```
  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*::*p*
hpx::*parallel*::*util*::detail::algorithm_result<*ExPolicy*, *hpx*::*ranges*::copy_if_result<**typename** *hpx*::*traits*::range_traits<*Rng*

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.

Header `hpx/parallel/container_algorithms/count.hpp`

`namespace hpx`

`namespace ranges`

Functions

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_

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.

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_

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:

```
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.

Header `hpx/parallel/container_algorithms/destroy.hpp`

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.

Header `hpx/parallel/container_algorithms/equal.hpp`

namespace hpx

namespace ranges

Functions

template<typename **ExPolicy**, typename **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pr
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:

```
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), *i 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. If the length of the range [first1, last1) does not equal the length of the range [first2, last2), it returns false.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1**
util::detail::algorithm_result<*ExPolicy*, bool>::type **equal**(*ExPolicy &&policy*, *Rng1 &&rng1*, *Rng2 &&rng2*, *Pred &&op* = *Pred*(), *Proj1 &&proj1* = *Proj1*(), *Proj2 &&proj2* = *Proj2*())

Returns true if the range [first1, last1) is equal to the range starting at first2, and false otherwise.

The comparison operations in the parallel *equal* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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), *i 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.

Header hpx/parallel/container_algorithms/fill.hpp

namespace hpx

Functions

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*.

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 *count* assignments, for count > 0.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.

- `Iterator`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `Size`: The type of the argument specifying the number of elements to apply *f* to.
- `T`: The type of the value to be assigned (deduced).

Parameters

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

Return The *fill_n* algorithm returns a *hpx::future<void>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *difference_type* otherwise (where *difference_type* is defined by *void*.

Header `hpx/parallel/container_algorithms/find.hpp`

`namespace hpx`

`namespace ranges`

Functions

template<typename **ExPolicy**, typename **Iter**, typename **Sent**, typename **T**, typename **Proj** = *util*::projection_identity
util::detail::algorithm_result<*ExPolicy*, *Iter*>::type **find**(*ExPolicy* &&*policy*, *Iter first*, *Sent last*, *T* **const** &*val*, *Proj* &&*proj* = *Proj*())

Returns the first element in the range [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most last - first applications of the operator==().

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Iter`: The type of the begin source iterators used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the end source iterators used (deduced). This iterator type must meet the requirements of an sentinel for Iter.
- `T`: The type of the value to find (deduced).
- `Proj`: The type of an optional projection function. This defaults to *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 the first range the algorithm will be applied to.

- `last`: Refers to the end of the sequence of elements of the first range the algorithm will be applied to.
- `val`: the value to compare the elements to
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

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 **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 [first, last) that is equal to value

The comparison operations in the parallel *find* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: At most last - first applications of the operator==().

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `T`: The type of the value to find (deduced).
- `Proj`: The type of an optional projection function. This defaults to *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.
- `val`: the value to compare the elements to
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

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 **Iter1**, typename **Sent1**, typename **Iter2**, typename **Sent2**, typename **Pr**

util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<Rng1>::type>::type **find_end**(*ExPolicy* &&*policy*, *Iter1 first1*, *Sent1 last1*, *Iter2 first2*, *Sent2 last2*, *Pred* &&*op* = *Pred*(), *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*())

Returns the last subsequence of elements [first2, last2) found in the range [first1, last1) using the given predicate *f* to compare elements.

The comparison operations in the parallel *find_end* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Iter1`: The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent1`: The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- `Iter2`: The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent2`: The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `Proj1`: The type of an optional projection function applied to the first sequence. This defaults to *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 first sequence of elements the algorithm will be applied to.
- `last1`: Refers to the end of the first sequence of elements the algorithm will be applied to.
- `first2`: Refers to the beginning of the second sequence of elements the algorithm will be

applied to.

- `last2`: Refers to the end of the second sequence of elements the algorithm will be applied to.
- `op`: The binary predicate which returns *true* if the elements should be treated as equal. The signature should be equivalent to the following:

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

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *iterator_t<Rng>* and *iterator_t<Rng2>* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively.
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of the first range of type dereferenced *iterator_t<Rng1>* as a projection operation before the function *op* is invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of the second range of type dereferenced *iterator_t<Rng2>* as a projection operation before the function *op* is invoked.

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 **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1**
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* = distance(begin(rng2), end(rng2)) and *N* = distance(begin(rng), end(rng)).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1`: The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- `Rng2`: The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `Proj1`: The type of an optional projection function applied to the first sequence. This defaults to *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:

```
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 **Pr**

util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<Rng1>::type>::type **find_first_of**(*Ex* *&& icy* *Iter* *firs* *Sen* *las* *Iter* *firs* *Sen* *las* *Pre* *&&* = *Pre* *Pro* *&&* = *Pro* *Pro* *&&* = *Pro*

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* = distance(first2, last2) and *N* = distance(first1, last1).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Iter1`: The type of the begin source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent1`: The type of the end source iterators for the first sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter1.
- `Iter2`: The type of the begin source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent2`: The type of the end source iterators for the second sequence used (deduced). This iterator type must meet the requirements of an sentinel for Iter2.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `Proj1`: The type of an optional projection function. This defaults to *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:

```
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.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1**
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng1*>::type>::type **find_first_of**(*Ex*
&&
icy,
Rng
&&
Rng
&&
Pre
&&
=
Pre
Pro
&&
=
Pro
Pro
&&
=
Pro

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* = distance(begin(rng2), end(rng2)) and *N* = distance(begin(rng1), end(rng1)).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1`: The type of the first source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- `Rng2`: The type of the second source range (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *replace* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `Proj1`: The type of an optional projection function. This defaults to *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:

```
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.

Header `hpx/parallel/container_algorithms/for_each.hpp`

namespace hpx

namespace ranges

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:

```
<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:

```
<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.

template<typename **ExPolicy**, typename **FwdIter**, typename **Sent**, typename **F**, typename **Proj** = *util*::projection_ide
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:

```
<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.

template<typename **Rng**, typename **F**, typename **Proj** = *util*::projection_identity>
hpx::*ranges*::for_each_result<**typename** *hpx*::*traits*::range_iterator<*Rng*>::type, *F*> **for_each** (ExPolicy *&&policy*, *Rng &&rng*, *F &&f*, *Proj &&proj* = *Proj*())

Applies *f* to the result of dereferencing every iterator in the given range *rng*.

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *size(rng)* times.

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

Return {std::end(rng), std::move(f)}

Template Parameters

- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *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

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). The signature of this predicate should be equivalent to:

```
<ignored> pred(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

template<typename **ExPolicy**, typename **Rng**, typename **F**, typename **Proj** = *util*::projection_identity>
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **for_each**(*ExPolicy* &&*policy*, *Rng* &&*rng*, *F* &&*f*, *Proj* &&*proj* = *Proj*())

Applies *f* to the result of dereferencing every iterator in the given range *rng*.

If *f* returns a result, the result is ignored.

Note Complexity: Applies *f* exactly *size(rng)* 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.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *for_each* requires *F* to meet the requirements of *CopyConstructible*.

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

```
<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.

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<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**, typename **Proj** = *util*::projection_id
util::detail::algorithm_result<*ExPolicy*, *FwdIter*>::type **for_each_n**(*ExPolicy* &&*policy*, *FwdIter first*, *Size count*, *F* &&*f*, *Proj* &&*proj* = *Proj*())

Applies *f* to the result of dereferencing every iterator in the range [first, first + count), starting from first and proceeding to first + count - 1.

If *f* returns a result, the result is ignored.

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.

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.

Header `hpx/parallel/container_algorithms/generate.hpp`

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 *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*.

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:

```
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*.

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 *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*.

Header `hpx/parallel/container_algorithms/is_heap.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **Comp** = detail::less, typename **Proj** = *util*::*projection_ide*
util::detail::algorithm_result<*ExPolicy*, bool>::type **is_heap**(*ExPolicy* &&*policy*, *Rng* &&*rng*, *Comp* &&*comp* = *Comp*(), *Proj* &&*proj* = *Proj*())

Returns whether the range is max heap. That is, true if the range is max heap, false otherwise. The function uses the given comparison function object *comp* (defaults to using operator<()).

comp has to induce a strict weak ordering on the values.

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.

template<typename **ExPolicy**, typename **Rng**, typename **Comp** = detail::less, typename **Proj** = *util*::*projection_ide*
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **is_heap_until** (

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 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_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.

Header hpx/parallel/container_algorithms/merge.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **RandIter3**, typename **Comp** = detail:
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in1 (**typename** *hpx*::*traits*::range_iterator<*Rng1*>::type), *tag*::in2 **typename** *hpx*::*traits*::range_iterator<*Rng2*>::type, *tag*::out*RandIter3*>>::type **merge***ExPolicy* &&*policy*, *Rng1* &&*rng1*, *Rng2* &&*rng2*, *RandIter3 dest*, *Comp* &&*comp* = *Comp*(), *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*()Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at *dest*. The order of equivalent elements in the each of original two ranges is preserved. For equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range. The destination range cannot overlap with either of the input ranges.

The assignments in the parallel *merge* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs O(std::distance(first1, last1) + std::distance(first2, last2)) applications of the comparison *comp* and the each projection.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1`: The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- `Rng2`: The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an random access iterator.
- `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<>
- `Proj1`: The type of an optional projection function to be used for elements of the first range. This defaults to 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.
- `rng1`: Refers to the first range of elements the algorithm will be applied to.
- `rng2`: Refers to the second range of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `comp`: *comp* is a callable object which returns true if the first argument is less than the second, and false otherwise. The signature of this comparison should be equivalent to:

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

The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *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<tagged_tuple<tag::in1(RandIter1), tag::in2(RandIter2), tag::out(RandIter3)> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_tuple<tag::in1(RandIter1), tag::in2(RandIter2), tag::out(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.

template<typename **ExPolicy**, typename **Rng**, typename **RandIter**, typename **Comp** = detail::less, typename **Pro**
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(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:

```
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*

Header `hpx/parallel/container_algorithms/minmax.hpp`

`namespace hpx`

`namespace parallel`

`namespace v1`

Functions

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 **min_elemen**

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.

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_elemen**

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 This argument is optional and defaults to std::less. the left argument is less than the right element. The signature of the predicate function should be equivalent to the following:

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

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

The comparisons in the parallel *max_element* algorithm invoked with an execution policy object of type *parallel_policy* or *parallel_task_policy* are permitted to execute in an unordered fashion in unspecified threads, and indeterminately sequenced within each thread.

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.

template<typename **ExPolicy**, typename **Rng**, typename **Proj** = *util*::*projection_identity*, typename **F** = detail::less>
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::min (**typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type) , *tag*::max **typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type>>::type **minmax_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 *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*(floor(3/2*(N-1)), 0) applications of the predicate, where N = std::distance(first, last).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *minmax_element* requires *F* to meet the requirements of *CopyConstructible*.
- `Proj`: The type of an optional projection function. This defaults to 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. This argument is optional and defaults to std::less. The signature of the predicate function should be equivalent to the following:

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

  The signature does not need to have const &, but the function must not modify the objects passed to it. The type *Type1* must be such that objects of type *FwdIter* can be dereferenced and then implicitly converted to *Type1*.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

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.

Header `hpx/parallel/container_algorithms/mismatch.hpp`

`namespace hpx`

`namespace ranges`

Functions

template<typename **`ExPolicy`**, typename **`Iter1`**, typename **`Sent1`**, typename **`Iter2`**, typename **`Sent2`**, typename **`Pr`**
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:

```
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.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = *ranges*::equal_to, typename **Proj1**

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:

```
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.

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).

Header `hpx/parallel/container_algorithms/move.hpp`

`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 *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.
- `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 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 *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*>

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 of the input iterator *last* and the output iterator to the element in the destination range, one past the last element moved.

Header hpx/parallel/container_algorithms/partition.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **partition** (*ExPolicy* &&*policy*, *Rng* &&*rng*, *Pred* &&*pred*, *Proj* &&*proj* = *Proj*())

Reorders the elements in the range *rng* in such a way that all elements for which the predicate *pred* returns true precede the elements for which the predicate *pred* returns false. Relative order of the elements is not preserved.

The assignments in the parallel *partition* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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:

```
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<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 **ExPolicy**, typename **Rng**, typename **FwdIter2**, typename **FwdIter3**, typename **Pred**, typen
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in (**typename** *hpx*::*traits*::range_iterator<*Rng*>::type), *tag*::out1
FwdIter2, *tag*::out2*FwdIter3*>>::type **partition_copy***ExPolicy* &&*policy*, *Rng* &&*rng*, *FwdIter2 dest_true*, *FwdIter3 dest_false*, *Pred* &&*pred*, *Proj* &&*proj* = *Proj*()Copies the elements in the range *rng*, to two different ranges depending on the value returned by the predicate *pred*. The elements, that satisfy the predicate *pred*, are copied to the range beginning at *dest_true*. The rest of the elements are copied to the range beginning at *dest_false*. The order of the elements is preserved.

The assignments in the parallel *partition_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than N assignments, exactly N applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range for the elements that satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- `FwdIter3`: The type of the iterator representing the destination range for the elements that don't satisfy the predicate *pred* (deduced). This iterator type must meet the requirements of an forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *partition_copy* requires *Pred* to meet the requirements of *CopyConstructible*.
- `Proj`: The type of an optional projection function. This defaults to 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_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 the range *rng*. This is an unary predicate for partitioning the source iterators. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *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.

Header `hpx/parallel/container_algorithms/reduce.hpp`

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(*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 iterator used (deduced). This iterator type must meet the requirements of an forward iterator.
- `Sent`: The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.
- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.

- `init`: The initial value for the generalized sum.

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 over 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 **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, …, *(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 requirements of an forward iterator.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.
- `init`: The initial value for the generalized sum.

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**, 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 an forward iterator.
- `Sent`: The type of the source sentinel used (deduced). This iterator type must meet the requirements of an forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `first`: Refers to the beginning of the sequence of elements the algorithm will be applied to.
- `last`: Refers to the end of the sequence of elements the algorithm will be applied to.

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 *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.

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, . . . , *(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.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.

- `F`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *copy_if* requires *F* to meet the requirements of *CopyConstructible*.
- `T`: The type of the value to be used as initial (and intermediate) values (deduced).

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type1 &b);
```

  The signature does not need to have const&. The types *Type1 Ret* must be such that an object of type *FwdIterB* can be dereferenced and then implicitly converted to any of those types.
- `init`: The initial value for the generalized sum.

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 over 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 **Rng**, typename **T**>
util::detail::algorithm_result<*ExPolicy*, *T*>::type **reduce**(*ExPolicy* &&*policy*, *Rng* &&*rng*, *T init*)

Returns GENERALIZED_SUM(+, init, *first, . . . , *(first + (last - first) - 1)).

The reduce operations in the parallel *reduce* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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 [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 **Rng**>
util::detail::algorithm_result<*ExPolicy*, **typename** *std*::iterator_traits<**typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_

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.
- `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 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 *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.

Header `hpx/parallel/container_algorithms/remove.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **T**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **remove**(*ExPolicy* &&*policy*, *Rng* &&*rng*, *T* **const** &*value*, *Proj* &&*proj* = *Proj*())

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==() 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<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *FwdIter* otherwise. The *remove* algorithm returns the iterator to the new end of the range.

template<typename **ExPolicy**, typename **Rng**, typename **Pred**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **remove_if**(*ExPol* *&&po* *icy*, *Rng* *&&rn* *Pred* *&&pr* *Proj* *&&pr* = *Proj*()

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* 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:

```
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<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.

Header `hpx/parallel/container_algorithms/remove_copy.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**, typename **T**, typename **Proj** = *util*::*projection_*
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<**typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type, *O*

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 val. 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(proj, *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.
- `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.
- `T`: The type that the result of dereferencing InIter is compared 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.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `val`: Value to be removed.

- `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_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<tagged_pair<tag::in(InIter), tag::out(OutIter)> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(InIter), tag::out(OutIter)>* otherwise. The *copy* algorithm returns the pair of the input iterator forwarded to the first element after the last in the input sequence and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**, typename **F**, typename **Proj** = *util*::*projection_*
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<**typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type, *O*

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, INVOKE(proj, *it)) != false.

The assignments in the parallel *remove_copy_if* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

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.
- `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

elements to be removed. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to Type.

- 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_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<tagged_pair<tag::in(InIter), tag::out(OutIter)> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(InIter), tag::out(OutIter)>* 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.

Header hpx/parallel/container_algorithms/replace.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **T1**, typename **T2**, typename **Proj** = *util*::*projection_ident*
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::type **replace** (*Ex*
&&
icy
Rn
&&
T1
co
&o
T2
co
&n
Pr
&&
=
Pr

Replaces all elements satisfying specific criteria with *new_value* in the range [first, last).

Effects: Substitutes elements referred by the iterator it in the range [first,last) with new_value, when the following corresponding conditions hold: INVOKE(proj, *i) == old_value

Note Complexity: Performs exactly *last* - *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.
- `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 *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 **ExPolicy**, typename **Rng**, typename **F**, typename **T**, typename **Proj** = *util*::*projection_identity*>
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type>::type **replace_if**

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, INVOKE(proj, *it)) != false

Note Complexity: Performs exactly *last* - *first* 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.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range 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*. (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.
- `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 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<FwdIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *void* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**, typename **T1**, typename **T2**, typename **Proj** =
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (**typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type), *tag*::out *OutIter*>>::type **replace_copy***ExPolicy* &&*policy*, *Rng* &&*rng*, *OutIter dest*, *T1* **const** &*old_value*, *T2* **const** &*new_value*, *Proj* &&*proj* = *Proj*()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(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 *last* - *first* applications of the predicate.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `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

- `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<tagged_pair<tag::in(InIter), tag::out(OutIter)> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(InIter), tag::out(OutIter)>* otherwise. The *copy* algorithm returns the pair of the input iterator *last* and the output iterator to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**, typename **F**, typename **T**, typename **Proj** = *ut*
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (**typename** *hpx*::*traits*::range_traits<*Rng*>::iterator_type) , *tag*::out
OutIter>>::type **replace_copy_if***ExPolicy* &&*policy*, *Rng* &&*rng*, *OutIter dest*, *F* &&*f*, *T* **const** &*new_value*, *Proj* &&*proj* = *Proj*()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, 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 *last* - *first* applications of the predicate.

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `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 *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.
- `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 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_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<tagged_pair<tag::in(InIter), tag::out(OutIter)> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *tagged_pair<tag::in(InIter), tag::out(OutIter)>* otherwise. 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.

Header `hpx/parallel/container_algorithms/reverse.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**>
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **reverse**(*ExPolicy* &&*policy*, *Rng* &&*rng*)

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<BidirIter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *BidirIter* otherwise. It returns *last*.

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**>

util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<**typename** *hpx*::*traits*::range_iterator<*Rng*>::type, *OutIter*>

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.
- `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.
- `dest_first`: 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<in_out_result<BidirIter, OutIter> >* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *in_out_result<BidirIter, OutIter>* 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.

Header `hpx/parallel/container_algorithms/rotate.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**>
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<**typename** *hpx*::*traits*::range_iterator<*Rng*>::type, **typena**

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.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of a forward iterator.

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `middle`: Refers to the element that should appear at the beginning of the rotated range.

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<tagged_pair<tag::begin(FwdIter), tag::end(FwdIter)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_pair<tag::begin(FwdIter), tag::end(FwdIter)>* otherwise. The *rotate* algorithm returns the iterator equal to pair(first + (last - new_first), last).

template<typename **ExPolicy**, typename **Rng**, typename **OutIter**>
util::detail::algorithm_result<*ExPolicy*, *util*::in_out_result<**typename** *hpx*::*traits*::range_iterator<*Rng*>::type, *OutIter*

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<tagged_pair<tag::in(FwdIter), tag::out(OutIter)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_pair<tag::in(FwdIter), tag::out(OutIter)>* otherwise. The *rotate_copy* algorithm returns the output iterator to the element past the last element copied.

Header `hpx/parallel/container_algorithms/search.hpp`

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = detail::equal_to, typename **Pro**
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng1*>::type>::type **search**(*ExPolicy* &&*policy*, *Rng1* &&*rng1*, *Rng2* &&*rng2*, *Pred* &&*op* = *Pred*(), *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*())

Searches the range [first, last) for any elements in the range [s_first, s_last). Uses a provided predicate to compare elements.

The comparison operations in the parallel *search* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: at most (S*N) comparisons where *S* = distance(s_first, s_last) and *N* = distance(first, last).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng1`: The type of the examine range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Rng2`: The type of the search range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Pred`: The type of an optional function/function object to use. Unlike its sequential form, the parallel overload of *adjacent_find* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `Proj1`: The type of an optional projection function. This defaults to 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

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the sequence of elements the algorithm will be examining.
- `rng2`: Refers to the sequence of elements the algorithm will be searching for.
- `op`: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

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

The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and

FwdIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively

- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

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.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **Pred** = detail::equal_to, typename **Pro**
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng1*>::type>::type **search_n** (*ExPol*
&&po
icy,
Rng1
&&rn
std::si
count,
Rng2
&&rn
Pred
&&op
=
Pred()
Proj1
&&pr
=
Proj1(
Proj2
&&pr
=
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.
- `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 *CopyCon-*

structible. 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
- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng1`: Refers to the sequence of elements the algorithm will be examining.
- `count`: The number of elements to apply the algorithm on.
- `rng2`: Refers to the sequence of elements the algorithm will be searching for.
- `op`: Refers to the binary predicate which returns true if the elements should be treated as equal. the signature of the function should be equivalent to

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

  The signature does not need to have const &, but the function must not modify the objects passed to it. The types *Type1* and *Type2* must be such that objects of types *FwdIter1* and *FwdIter2* can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively
- `proj1`: Specifies the function (or function object) which will be invoked for each of the elements of *rng1* as a projection operation before the actual predicate *is* invoked.
- `proj2`: Specifies the function (or function object) which will be invoked for each of the elements of *rng2* as a projection operation before the actual predicate *is* invoked.

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.

Header `hpx/parallel/container_algorithms/sort.hpp`

namespace hpx

namespace parallel

namespace rangev1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **Compare** = *v1*::detail::less, typename **Proj** = *util*::*projec*

util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **sort** (*ExPolicy* &&*policy*, *Rng* &&*rng*, *Compare* &&*comp* = *Compare*(), *Proj* &&*proj* = *Proj*())

Sorts the elements in the range *rng* in ascending order. The order of equal elements is not guaranteed to be preserved. The function uses the given comparison function object comp (defaults to using operator<()).

A sequence is sorted with respect to a comparator *comp* and a projection *proj* if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, and INVOKE(comp, INVOKE(proj, *(i + n)), INVOKE(proj, *i)) == false.

Note Complexity: O(Nlog(N)), where N = 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<Iter>* if the execution policy is of type *sequenced_task_policy* or *parallel_task_policy* and returns *Iter* otherwise. It returns *last*.

Header `hpx/parallel/container_algorithms/stable_sort.hpp`

namespace hpx

namespace parallel

namespace rangev1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **Compare** = *v1*::detail::less, typename **Proj** = *util*::*projec*
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **stable_sort** (*Ex*
&&
icy
Rn
&&
Co
pa
&&
=
Co
pa
Pr
&&
=
Pr

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 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 *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.

Header `hpx/parallel/container_algorithms/transform.hpp`

`namespace hpx`

`namespace parallel`

`namespace v1`

Functions

template<typename **`ExPolicy`**, typename **`Rng`**, typename **`OutIter`**, typename **`F`**, typename **`Proj`** = *util*::*projection_*
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (**`typename`** *hpx*::*traits*::range_iterator<*Rng*>::type) , *tag*::out *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 *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*.
- `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 *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.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.

- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is an unary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type &a);
```

  The signature does not need to have const&. The type *Type* must be such that an object of type *InIter* can be dereferenced and then implicitly converted to *Type*. The type *Ret* must be such that an object of type *OutIter* can be dereferenced and assigned a value of type *Ret*.
- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *f* is invoked.

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<tagged_pair<tag::in(InIter), tag::out(OutIter)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_pair<tag::in(InIter), tag::out(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 **Rng**, typename **InIter2**, typename **OutIter**, typename **F**, typename **P**
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in1 (**typename** *hpx*::*traits*::range_iterator<*Rng*>::type), *tag*::in2 *InIter2*, *tag*::out*OutIter*>>::type **transform***ExPolicy* &&*policy*, *Rng* &&*rng*, *InIter2 first2*, *OutIter dest*, *F* &&*f*, *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*()Applies the given function *f* to pairs of elements from two ranges: one defined by *rng* and the other beginning at first2, and stores the result in another range, beginning at dest.

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*.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `InIter2`: The type of the source iterators for the second range 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.
- `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.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `first2`: Refers to the beginning of the second sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements

in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types InIter1 and InIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *OutIter* 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<tagged_tuple<tag::in1(InIter1), tag::in2(InIter2), tag::out(OutIter)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_tuple<tag::in1(InIter1), tag::in2(InIter2), tag::out(OutIter)>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element after the first input sequence, an iterator referring to the first element after the second input sequence, and the output iterator referring to the element in the destination range, one past the last element copied.

template<typename **ExPolicy**, typename **Rng1**, typename **Rng2**, typename **OutIter**, typename **F**, typename **Pro**
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_tuple<*tag*::in1 (**typename** *hpx*::*traits*::range_iterator<*Rng1*>::type), *tag*::in2
typename *hpx*::*traits*::range_iterator<*Rng2*>::type, *tag*::out*OutIter*>>::type
transform*ExPolicy* &&*policy*, *Rng1* &&*rng1*, *Rng2* &&*rng2*, *OutIter dest*, *F* &&*f*, *Proj1* &&*proj1* = *Proj1*(), *Proj2* &&*proj2* = *Proj2*()Applies the given function *f* to pairs of elements from two ranges: one defined by [first1, last1) and the other beginning at first2, and stores the result in another range, beginning at dest.

The invocations of *f* in the parallel *transform* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Exactly min(last2-first2, last1-first1) 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*.
- `Rng1`: The type of the first source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `Rng2`: The type of the second source range used (deduced). The iterators extracted from this range type must meet the requirements of an input iterator.
- `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 *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.
- `rng1`: Refers to the first sequence of elements the algorithm will be applied to.
- `rng2`: Refers to the second sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `f`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last).This is a binary predicate. The signature of this predicate should be equivalent to:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&. The types *Type1* and *Type2* must be such that objects of types InIter1 and InIter2 can be dereferenced and then implicitly converted to *Type1* and *Type2* respectively. The type *Ret* must be such that an object of type *OutIter* 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.

Note The algorithm will invoke the binary predicate until it reaches the end of the shorter of the two given input sequences

Return The *transform* algorithm returns a *hpx::future<tagged_tuple<tag::in1(InIter1), tag::in2(InIter2), tag::out(OutIter)> >* if the execution policy is of type *parallel_task_policy* and returns *tagged_tuple<tag::in1(InIter1), tag::in2(InIter2), tag::out(OutIter)>* otherwise. The *transform* algorithm returns a tuple holding an iterator referring to the first element r 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.

Header `hpx/parallel/container_algorithms/transform_reduce.hpp`

namespace hpx

Functions

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(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.
- `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:

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The types *Type1*, *Type2*, and *Ret* must be such that an object of a type as returned from *conv_op* can be implicitly converted to any of those types.
- `conv_op`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by [first, last). This is a unary predicate. The signature of this predicate should be equivalent to:

  ```
  R fun(const Type &a);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *Iter* can be dereferenced and then implicitly converted to Type. The type *R* must be such that an object of this type can be implicitly converted to *T*.

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 **Rng1**, typename **FwdIter2**, typename **T**>
util::detail::algorithm_result<*ExPolicy*, *T*>::type **transform_reduce**(*ExPolicy &&policy*, *Rng1 &&rng1*, *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.
- `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).

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.

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.

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 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.
- `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:

  ```
  Ret fun(const Type1 &a, const Type1 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to a type of *T*.
- `conv_op`: Specifies the function (or function object) which will be invoked for each of the input values of the sequence. This is a binary predicate. The signature of this predicate should be equivalent to

  ```
  Ret fun(const Type1 &a, const Type2 &b);
  ```

  The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Ret* must be such that it can be implicitly converted to an object for the second argument type of *op1*.

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.

Header hpx/parallel/container_algorithms/unique.hpp

namespace hpx

namespace parallel

namespace v1

Functions

template<typename **ExPolicy**, typename **Rng**, typename **Pred** = detail::equal_to, typename **Proj** = *util*::*projection*
util::detail::algorithm_result<*ExPolicy*, **typename** *hpx*::*traits*::range_iterator<*Rng*>::type>::type **unique** (*ExPolicy* *&&policy*, *Rng* *&&rng*, *Pred* *&&pred* = *Pred*(), *Proj* *&&proj* = *Proj*())

Eliminates all but the first element from every consecutive group of equivalent elements from the range *rng* and returns a past-the-end iterator for the new logical end of the range.

The assignments in the parallel *unique* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred* and no more than twice as many applications of the projection *proj*, where N = std::distance(begin(rng), end(rng)).

Template Parameters
- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `Pred`: The type of the function/function object to use (deduced). Unlike its sequential form, the parallel overload of *unique* requires *Pred* to meet the requirements of *CopyConstructible*. This defaults to std::equal_to<>
- `Proj`: The type of an optional projection function. This defaults to 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 binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

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.

template<typename **ExPolicy**, typename **Rng**, typename **FwdIter2**, typename **Pred** = detail::equal_to, typename
util::detail::algorithm_result<*ExPolicy*, *hpx*::*util*::tagged_pair<*tag*::in (**typename** *hpx*::*traits*::range_iterator<*Rng*>::type), *tag*::out
FwdIter2>>::type **unique_copy**(*ExPolicy* &&*policy*, *Rng* &&*rng*, *FwdIter2 dest*, *Pred* &&*pred* = *Pred*(), *Proj* &&*proj* = *Proj*())Copies the elements from the range *rng*, to another range beginning at *dest* in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

The assignments in the parallel *unique_copy* algorithm invoked with an execution policy object of type *sequenced_policy* execute in sequential order in the calling thread.

Note Complexity: Performs not more than N assignments, exactly N - 1 applications of the predicate *pred*, where N = std::distance(begin(rng), end(rng)).

Template Parameters

- `ExPolicy`: The type of the execution policy to use (deduced). It describes the manner in which the execution of the algorithm may be parallelized and the manner in which it executes the assignments.
- `Rng`: The type of the source range used (deduced). The iterators extracted from this range type must meet the requirements of an forward iterator.
- `FwdIter2`: The type of the iterator representing the destination range (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

Parameters

- `policy`: The execution policy to use for the scheduling of the iterations.
- `rng`: Refers to the sequence of elements the algorithm will be applied to.
- `dest`: Refers to the beginning of the destination range.
- `pred`: Specifies the function (or function object) which will be invoked for each of the elements in the sequence specified by the range *rng*. This is an binary predicate which returns *true* for the required elements. The signature of this predicate should be equivalent to:

```
bool pred(const Type &a, const Type &b);
```

The signature does not need to have const&, but the function must not modify the objects passed to it. The type *Type* must be such that an object of type *FwdIter1* can be dereferenced and then implicitly converted to *Type*.

- `proj`: Specifies the function (or function object) which will be invoked for each of the elements as a projection operation before the actual predicate *is* invoked.

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<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 *unique_copy* algorithm returns the pair of the source iterator to *last*, and the destination iterator to the end of the *dest* range.

Header `hpx/parallel/container_memory.hpp`

Header `hpx/parallel/container_numeric.hpp`

Header `hpx/parallel/datapar.hpp`

Header `hpx/parallel/datapar/iterator_helpers.hpp`

Header `hpx/parallel/datapar/loop.hpp`

Header `hpx/parallel/datapar/transform_loop.hpp`

Header `hpx/parallel/datapar/zip_iterator.hpp`

Header `hpx/parallel/memory.hpp`

Header `hpx/parallel/numeric.hpp`

Header `hpx/parallel/spmd_block.hpp`

namespace hpx

namespace lcos

namespace local

Functions

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

template<typename **F**, typename ...**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*) **const**

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

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

namespace v2

Typedefs

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

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

template<typename **F**, typename ...**Args**>
void **define_spmd_block** (*std*::size_t *num_images*, *F &&f*, *Args&&... args*)

Header `hpx/parallel/tagspec.hpp`

Header `hpx/parallel/task_block.hpp`

namespace hpx

namespace parallel

namespace v2

Functions

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

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*).

template<typename **ExPolicy** = *parallel*::*execution*::*parallel_policy*>
class task_block

#include <task_block.hpp> The class *task_block* defines an interface for forking and joining parallel tasks. The *define_task_block* and *define_task_block_restore_thread* function templates create an object of type *task_block* and pass a reference to that object to a user-provided callable object.

An object of class task_block cannot be constructed, destroyed, copied, or moved except by the implementation of the task region library. Taking the address of a *task_block* object via operator& or addressof is ill formed. The result of obtaining its address by any other means is unspecified.

A task_block is active if it was created by the nearest enclosing task block, where "task block" refers to an invocation of define_task_block or define_task_block_restore_thread and "nearest

enclosing" means the most recent invocation that has not yet completed. Code designated for execution in another thread by means other than the facilities in this section (e.g., using thread or async) are not enclosed in the task region and a *task_block* passed to (or captured by) such code is not active within that code. Performing any operation on a *task_block* that is not active results in undefined behavior.

The task_block that is active before a specific call to the run member function is not active within the asynchronous function that invoked run. (The invoked function should not, therefore, capture the task_block from the surrounding block.)

```
Example:
    define_task_block([&](auto& tr) {
        tr.run([&] {
```

(continues on next page)

(continued from previous page)

```
            tr.run([] { f(); });                  // Error: tr is not␣
↪active
            define_task_block([&](auto& tr) {    // Nested task block
                tr.run(f);                        // OK: inner tr is␣
↪active
                /// ...
            });
        });
        /// ...
    });
```

Template Parameters

- `ExPolicy`: The execution policy an instance of a task_block was created with. This defaults to *parallel_policy*.

Public Types

typedef ExPolicy **execution_policy**
Refers to the type of the execution policy used to create the task_block.

Public Functions

execution_policy **const** &**get_execution_policy**() **const**
Return the execution policy instance used to create this task_block

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.

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.

void **wait**()

Blocks until the tasks spawned using this *task_block* have finished.

Precondition: this shall be the active *task_block*.

Postcondition: All tasks spawned by the nearest enclosing task region have finished. A call to wait may return on a different thread than that on which it was called.

```
Example:
    define_task_block([&](auto& tr) {
        tr.run([&]{ process(a, w, x); }); // Process a[w] through␣
↪a[x]
        if (y < x) tr.wait();    // Wait if overlap between [w, x)␣
↪and [y, z)
        process(a, y, z);        // Process a[y] through a[z]
    });
```

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

Exceptions

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

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

mutex_type **mtx_**

std::vector<*hpx*::future<void>> **tasks_**

parallel::exception_list **errors_**

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()

Header `hpx/parallel/util/cancellation_token.hpp`

namespace hpx

namespace parallel

namespace util

template<typename **T** = detail::no_data, typename **Pred** = *std*::less_equal<*T*>>
class cancellation_token

#include <cancellation_token.hpp>

Public Functions

cancellation_token(T *data*)

bool **was_cancelled**(T *data*) **const**

void **cancel**(T *data*)

T **get_data**() **const**

Private Types

typedef *std*::atomic<T> **flag_type**

Private Members

std::shared_ptr<*flag_type*> **was_cancelled_**

Header `hpx/parallel/util/compare_projected.hpp`

template<typename **Compare**>
struct compare_projected<*Compare*, *util*::*projection_identity*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**>
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*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**, typename **Proj2_**>
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*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**, typename **Proj1_**>
compare_projected(*Compare_ &&comp, Proj1_ &&proj1, util::projection_identity*)

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, util::projection_identity*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**>
compare_projected(*Compare_ &&comp, util::projection_identity, util::projection_identity*)

template<typename **T1**, typename **T2**>
constexpr bool **operator()**(*T1 &&t1, T2 &&t2*) **const**

Public Members

Compare **comp_**

namespace hpx

namespace parallel

namespace util

template<typename **Compare**, typename **Proj**>
struct compare_projected<*Compare, Proj*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**, typename **Proj_**>
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*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**, typename **Proj1_**, typename **Proj2_**>
compare_projected(*Compare_ &&comp*, *Proj1_ &&proj1*, *Proj2_ &&proj2*)

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*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**, typename **Proj1_**>
compare_projected(*Compare_ &&comp*, *Proj1_ &&proj1*, *util::projection_identity*)

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*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**>
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*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**, typename **Proj2_**>
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**>
struct compare_projected<*Compare*, *util::projection_identity*, *util::projection_identity*>
#include <compare_projected.hpp>

Public Functions

template<typename **Compare_**>
compare_projected(*Compare_ &&comp*, *util::projection_identity*, *util::projection_identity*)

template<typename **T1**, typename **T2**>
constexpr bool **operator()**(*T1 &&t1*, *T2 &&t2*) **const**

Public Members

Compare **comp_**

Header hpx/parallel/util/foreach_partitioner.hpp

Header hpx/parallel/util/invoke_projected.hpp

namespace hpx

namespace parallel

namespace util

template<typename **Pred**, typename **Proj**>
struct invoke_projected
#include <invoke_projected.hpp>

Public Types

typedef *hpx*::*util*::decay<Pred>::type **pred_type**

typedef *hpx*::*util*::decay<Proj>::type **proj_type**

Public Functions

template<typename **Pred_**, typename **Proj_**>
invoke_projected(*Pred_ &&pred*, *Proj_ &&proj*)

template<typename **T**>
auto **operator()**(*T &&t*)

Public Members

pred_type **pred_**

proj_type **proj_**

Header hpx/parallel/util/loop.hpp

namespace hpx

namespace parallel

namespace util

Functions

template<typename **ExPolicy**, typename **VecOnly**, typename **F**, typename ...**Iters**>
std::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, **typename** *hpx*::*util*::invoke_result<*F*, *I*

template<typename **ExPolicy**, typename **Iter**>
constexpr *std*::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, bool>::type **loop_optin**

template<typename **ExPolicy**, typename **Begin**, typename **End**, typename **F**>
constexpr *Begin* **loop** (*ExPolicy*&&, *Begin begin*, *End end*, *F* &&*f*)

template<typename **ExPolicy**, typename **Begin**, typename **End**, typename **CancelToken**, typename **F**>
constexpr *Begin* **loop** (*ExPolicy*&&, *Begin begin*, *End end*, *CancelToken* &*tok*, *F* &&*f*)

template<typename **ExPolicy**, typename **VecOnly**, typename **Begin1**, typename **End1**, typename **Begin2**, typen
constexpr *std*::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, *std*::pair<*Begin1*, *Begin2*>

template<typename **ExPolicy**, typename **Iter**, typename **F**>
constexpr *std*::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, *Iter*>::type **loop_n** (*Iter it*, *std*:: *coun* *F* &&*f*

template<typename **ExPolicy**, typename **Iter**, typename **CancelToken**, typename **F**>
constexpr *std*::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, *Iter*>::type **loop_n** (*Iter it*, *std*:: *coun* *Cancel-Token* &*tok* *F* &&*f*

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*)

template<typename **Iter**, typename **T**, typename **Pred**>
T **accumulate_n** (*Iter it*, *std*::size_t *count*, *T init*, *Pred &&f*)

template<typename **T**, typename **Iter**, typename **Reduce**, typename **Conv** = *util*::*projection_identity*>
T **accumulate** (*Iter first*, *Iter last*, *Reduce &&r*, *Conv &&conv* = *Conv*())

template<typename **T**, typename **Iter1**, typename **Iter2**, typename **Reduce**, typename **Conv**>
T **accumulate** (*Iter1 first1*, *Iter1 last1*, *Iter2 first2*, *Reduce &&r*, *Conv &&conv*)

Header `hpx/parallel/util/low_level.hpp`

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

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

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

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

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

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] last22`: : 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 **Co**
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

-

Header `hpx/parallel/util/merge_four.hpp`

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** (*util*::*range*<*Iter1*, *Sent1*> &*rdest*, *util*::*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**>
util::*range*<*Value**> **uninit_full_merge4** (*util*::*range*<*Value**> **const** &*dest*, *util*::*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

Header `hpx/parallel/util/merge_vector.hpp`

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
-

Header `hpx/parallel/util/nbits.hpp`

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*::uint32_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 std::uint32_t const hpx::parallel::util::tmsb[256]

Header `hpx/parallel/util/partitioner.hpp`

Header `hpx/parallel/util/partitioner_with_cleanup.hpp`

Header `hpx/parallel/util/prefetching.hpp`

namespace hpx

namespace parallel

namespace util

Functions

template<typename **Itr**, typename ...**Ts**>
detail::prefetcher_context<*Itr*, *Ts* **const**...> **make_prefetcher_context**(*Itr base_begin*, *Itr base_end*, *std*::size_t *p_factor*, *Ts* **const**&... *rngs*)

Header `hpx/parallel/util/projection_identity.hpp`

namespace hpx

namespace parallel

namespace util

struct projection_identity
#include <projection_identity.hpp>

Public Functions

template<typename **T**>
constexpr *T* &&**operator()**(*T* &&*val*) **const**

Header `hpx/parallel/util/range.hpp`

namespace hpx

namespace parallel

namespace util

Typedefs

using range = *hpx*::*util*::iterator_range<Iterator, Sentinel>

Functions

template<typename **Iter**, typename **Sent**>
range<*Iter*, *Sent*> **concat**(*range*<*Iter*, *Sent*> **const** &*it1*, *range*<*Iter*, *Sent*> **const** &*it2*)

concatenate two contiguous ranges

Return range resulting of the concatenation

Parameters

- `[in] it1:` : first range
- `[in] it2:` : second range

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

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

template<typename **Iter**, typename **Sent**>
void **destroy_range**(*range*<*Iter*, *Sent*> *r*)

destroy a range of objects

Parameters

- `[in] r`: : range to destroy

template<typename **Iter**, typename **Sent**>
range<*Iter*, *Sent*> **init**(*range*<*Iter*, *Sent*> **const** &*r*, **typename** *std*::iterator_traits<*Iter*>::value_type &*val*)

initialize a range of objects with the object val moving across them

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 **Se**
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 lements 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 **Value**, typename **Co**
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. Initially is uninitialize memory
- `[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 **Compare**>
range<*Iter2*, *Sent2*> **half_merge**(*range*<*Iter2*, *Sent2*> **const** &*dest*, *range*<*Iter1*, *Sent1*> **const** &*src1*, *range*<*Iter2*, *Sent2*> **const** &*src2*, *Compare comp*)

: Merge two buffers. The first buffer is in a separate memory

Return : range with the two buffers merged

Parameters

- `[in] dest`: : range where finish the two buffers merged
- `[in] src1`: : first range to merge in a separate memory
- `[in] src2`: : second range to merge, in the final part of the range where deposit the final results
- `[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 **Se**
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] src1`: : 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

-

Header hpx/parallel/util/result_types.hpp

namespace hpx

namespace parallel

namespace util

template<typename **I**, typename **F**>
struct in_fun_result
#include <result_types.hpp>

Public Functions

template<typename **I2**, typename **F2**, typename **Enable** = **typename** *std*::enable_if<*std*::is_convertible<I **con**
constexpr operator in_fun_result<I2, F2>() **const** &

template<typename **I2**, typename **F2**, typename **Enable** = **typename** *std*::enable_if<*std*::is_convertible<I, *I2*>:
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

template<typename **I1**, typename **I2**>
struct in_in_result
#include <result_types.hpp>

Public Functions

template<typename **II1**, typename **II2**, typename **Enable** = **typename** *std*::enable_if<*std*::is_convertible<I1
constexpr operator in_in_result<II1, II2>() **const** &

template<typename **II1**, typename **II2**, typename **Enable** = **typename** *std*::enable_if<*std*::is_convertible<I1,
constexpr operator in_in_result<II1, II2>() &&

template<typename **Archive**>
void **serialize**(*Archive &ar*, unsigned)

Public Members

HPX_NO_UNIQUE_ADDRESS I1 hpx::parallel::util::in_in_result::in1

HPX_NO_UNIQUE_ADDRESS I2 hpx::parallel::util::in_in_result::in2

template<typename **I**, typename **O**>
struct in_out_result
#include <result_types.hpp>

Public Functions

template<typename **I2**, typename **O2**, typename **Enable** = **typename** *std*::enable_if<*std*::is_convertible<I **con**
constexpr operator in_out_result<I2, O2>() **const** &

template<typename **I2**, typename **O2**, typename **Enable** = **typename** *std*::enable_if<*std*::is_convertible<I, *I2*>:
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

Header `hpx/parallel/util/scan_partitioner.hpp`

Header `hpx/parallel/util/tagged_pair.hpp`

namespace hpx

namespace util

Functions

template<typename **Tag1**, typename **Tag2**, typename **T1**, typename **T2**>
hpx::future<tagged_pair<*Tag1* (**typename** decay<*T1*>::type), *Tag2*
typename 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* (**typename** tuple_element<0, tuple<*Ts*...>>::type), *Tag2*
typename tuple_element<1, tuple<*Ts*...>>::type>> **make_tagged_pair***hpx*::future<tuple<*Ts*...>>
&&*f*

Header `hpx/parallel/util/tagged_tuple.hpp`

namespace hpx

namespace util

Functions

template<typename ...**Tags**, typename ...**Ts**>
hpx::future<**typename** detail::tagged_tuple_helper<tuple<*Ts*...>, **typename** *util*::make_index_pack<sizeof...(*Tags*)>::typ

Header `hpx/parallel/util/transfer.hpp`

namespace hpx

namespace parallel

namespace util

Functions

template<typename **InIter**, typename **Sent**, typename **OutIter**>
in_out_result<*InIter*, *OutIter*> **copy** (*InIter first*, *Sent last*, *OutIter dest*)

template<typename **InIter**, typename **OutIter**>
in_out_result<*InIter*, *OutIter*> **copy_n** (*InIter first*, *std*::size_t *count*, *OutIter dest*)

template<typename **InIter**, typename **OutIter**>
void **copy_synchronize** (*InIter* **const** &*first*, *OutIter* **const** &*dest*)

template<typename **InIter**, typename **Sent**, typename **OutIter**>
in_out_result<*InIter*, *OutIter*> **move** (*InIter first*, *Sent last*, *OutIter dest*)

template<typename **InIter**, typename **OutIter**>
in_out_result<*InIter*, *OutIter*> **move_n** (*InIter first*, *std*::size_t *count*, *OutIter dest*)

Header `hpx/parallel/util/transform_loop.hpp`

namespace hpx

namespace parallel

namespace util

Functions

template<typename **ExPolicy**, typename **Iter**, typename **OutIter**, typename **F**>
std::pair<*Iter*, *OutIter*> **transform_loop**(*ExPolicy*&&, *Iter it*, *Iter end*, *OutIter dest*, *F* &&*f*)

template<typename **ExPolicy**, typename **InIter1**, typename **InIter2**, typename **OutIter**, typename **F**>
std::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, *hpx*::*util*::tuple<*InIter1*, *InIter2*, *OutIter*

template<typename **ExPolicy**, typename **InIter1**, typename **InIter2**, typename **OutIter**, typename **F**>
std::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, *hpx*::*util*::tuple<*InIter1*, *InIter2*, *OutIter*

template<typename **ExPolicy**, typename **Iter**, typename **OutIter**, typename **F**>
std::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, *std*::pair<*Iter*, *OutIter*>>::type **transf**

template<typename **ExPolicy**, typename **InIter1**, typename **InIter2**, typename **OutIter**, typename **F**>

std::enable_if<!*execution*::is_vectorpack_execution_policy<*ExPolicy*>::value, *hpx*::*util*::tuple<*InIter1*, *InIter2*, *OutIter*

Header `hpx/parallel/util/zip_iterator.hpp`

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.

Header `hpx/allocator_support/allocator_deleter.hpp`

`namespace hpx`

`namespace util`

template<typename **`Allocator`**>
`struct allocator_deleter`
#include <allocator_deleter.hpp>

Public Functions

template<typename **`SharedState`**>
void **`operator()`**(*SharedState* **state*)

Public Members

Allocator **`alloc_`**

Header `hpx/allocator_support/internal_allocator.hpp`

namespace hpx

namespace util

Typedefs

using internal_allocator = *std*::allocator<T>

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.

Header `hpx/asio/asio_util.hpp`

namespace hpx

namespace util

Functions

bool **split_ip_address**(*std*::string **const** &*v*, *std*::string &*host*, *std*::uint16_t &*port*)

Header `hpx/asio/map_hostnames.hpp`

namespace hpx

namespace util

struct map_hostnames

#include <map_hostnames.hpp>

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.

Header `hpx/assert.hpp`

Header `hpx/assertion/current_function.hpp`

Defines

HPX_ASSERT_CURRENT_FUNCTION

Header hpx/assertion/evaluate_assert.hpp

Header hpx/assertion/source_location.hpp

namespace hpx

namespace assertion

Functions

std::ostream &**operator<<**(*std*::ostream &*os*, *source_location* **const** &*loc*)

struct source_location
#include <source_location.hpp> This contains the location information where *HPX_ASSERT* has been called

Public Members

const char ***file_name**

unsigned **line_number**

const char ***function_name**

Header hpx/modules/assertion.hpp

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*

namespace hpx

namespace assertion

Typedefs

using assertion_handler = void (*) (*source_location* **const** &loc, **const** char *expr, *std*::string **const** &msg)

The signature for an assertion handler.

Functions

void **set_assertion_handler** (*assertion_handler handler*)

Set the assertion handler to be used within a program. If the handler has been set already once, the call to this function will be ignored.

Note This function is not thread safe

async_base

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.

Header `hpx/async_base/apply.hpp`

namespace hpx

Functions

template<typename **F**, typename ...**Ts**>
bool **apply** (*F* &&*f*, *Ts*&&... *ts*)

Header `hpx/async_base/async.hpp`

namespace hpx

Functions

template<typename **F**, typename ...**Ts**>
decltype(auto) **async** (*F* &&*f*, *Ts*&&... *ts*)

Header `hpx/async_base/dataflow.hpp`

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*)

Header `hpx/async_base/launch_policy.hpp`

namespace hpx

struct launch : **public** detail::policy_holder<>
#include <launch_policy.hpp> Launch policies for hpx::async etc.

Public Functions

constexpr launch()
Default constructor. This creates a launch policy representing all possible launch modes

Public Static Attributes

const detail::fork_policy **fork**
Predefined launch policy representing asynchronous execution.The new thread is executed in a preferred way

const detail::sync_policy **sync**
Predefined launch policy representing synchronous execution.

const detail::deferred_policy **deferred**
Predefined launch policy representing deferred execution.

const detail::apply_policy **apply**
Predefined launch policy representing fire and forget execution.

const detail::select_policy_generator **select**
Predefined launch policy representing delayed policy selection.

Header `hpx/async_base/sync.hpp`

namespace hpx

Functions

template<typename **F**, typename ...**Ts**>
auto **sync**(*F* &&*f*, *Ts*&&... *ts*)

Header `hpx/async_base/traits/is_launch_policy.hpp`

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.

Header `hpx/async_combinators/split_future.hpp`

namespace hpx

Functions

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:

```
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::util::tuple<>*, *std::pair<T1, T2>*, and *std::array<T, N>*

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

Header `hpx/async_combinators/wait_all.hpp`

`namespace hpx`

Functions

template<typename **InputIter**>
void **wait_all** (*InputIter first*, *InputIter last*)

The function *wait_all* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note The function *wait_all* returns after all futures have become ready. All input futures are still valid after *wait_all* returns.

Parameters

- `first`: The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *wait_all* should wait.
- `last`: The iterator pointing to the last element of a sequence of *future* or *shared_future* objects for which *wait_all* should wait.

template<typename **R**>
void **wait_all** (*std*::vector<future<*R*>> &&*futures*)

The function *wait_all* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns after they finished executing.

Note The function *wait_all* returns after all futures have become ready. All input futures are still valid after *wait_all* returns.

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.

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.

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.

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*.

Header `hpx/async_combinators/wait_any.hpp`

namespace hpx

Functions

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.

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.

template<typename R, std:;size_t N>void hpx::wait_any(std::array< future< R >, N > & f

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] Amn array 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.

template<typename ...**T**>
void **wait_any**(*error_code* &*ec*, *T*&&... *futures*)

The function *wait_any* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function *wait_any* returns after at least one future has become ready. All input futures are still valid after *wait_any* returns.

Note 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 *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.

template<typename ...**T**>
void **wait_any**(*T&&... futures*)

The function *wait_any* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function *wait_any* returns after at least one future has become ready. All input futures are still valid after *wait_any* returns.

Note 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 *wait_any* should wait.

template<typename **InputIter**>
InputIter **wait_any_n**(*InputIter first*, *std*::size_t *count*, *error_code &ec = throws*)

The function *wait_any_n* is a non-deterministic choice operator. It OR-composes all future objects given and returns after one future of that list finishes execution.

Note The function *wait_any_n* returns after at least one future has become ready. All input futures are still valid after *wait_any_n* returns.

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 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_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.

Header `hpx/async_combinators/wait_each.hpp`

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.

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*.

Header `hpx/async_combinators/wait_some.hpp`

namespace hpx

Functions

template<typename **InputIter**>
future<vector<future<**typename** *std*::iterator_traits<*InputIter*>::value_type>>> **wait_some** (*std*::size_t *n*, Iterator *first*, Iterator *last*, error_code *&ec* = *throws*)

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<R>>>: 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.

template<typename **R**>
void **wait_some** (*std*::size_t *n*, *std*::vector<future<*R*>> &&*futures*, *error_code* &*ec* = *throws*)

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The function *wait_all* returns after *n* futures have become ready. All input futures are still valid after *wait_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] A vector 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.

template<typename **R**, *std*::size_t **N**>
void **wait_some** (*std*::size_t *n*, *std*::array<future<*R*>, *N*> &&*futures*, *error_code* &*ec* = *throws*)

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The function *wait_all* returns after *n* futures have become ready. All input futures are still valid after *wait_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 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.

template<typename ...**T**>
void **wait_some** (*std*::size_t *n*, *T*&&... *futures*, *error_code* &*ec* = *throws*)

The function *wait_some* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The function *wait_all* returns after *n* futures have become ready. All input futures are still valid after *wait_all* returns.

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.
- `futures`: [in] An arbitrary number of *future* or *shared_future* objects, possibly holding different types for which *wait_some* should wait.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

template<typename **InputIter**>
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.

Header `hpx/async_combinators/when_all.hpp`

`namespace hpx`

Functions

template<typename **`InputIter`**, typename **`Container`** = vector<future<**`typename`** *std*::iterator_traits<*InputIter*>::value_typ
future<*Container*> **`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_typ
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.

Header `hpx/async_combinators/when_any.hpp`

`namespace hpx`

Functions

template<typename **InputIter**, typename **Container** = vector<future<**typename** *std*::iterator_traits<*InputIter*>::value_typ
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<typename **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 *futures* or *shared_future* objects for which *when_any* should wait.

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.

template<typename **InputIter**, typename **Container** = vector<future<**typename** *std*::iterator_traits<*InputIter*>::value_typ
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*.

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.

Header `hpx/async_combinators/when_each.hpp`

namespace hpx

Functions

template<typename **F**, typename **Future**>
future<void> **when_each** (*F* &&*f*, *std*::vector<*Future*> &&*futures*)

The function *when_each* is an operator allowing to join on the results of all given futures. It AND-composes all future objects given and returns a new future object representing the event of all those futures having finished executing. It also calls the supplied callback for each of the futures which becomes ready.

Note This function consumes the futures as they are passed on to the supplied function. The callback should take one or two parameters, namely either a *future* to be processed or a type that *std::size_t* is implicitly convertible to as the first parameter and the *future* as the second parameter. The first parameter will correspond to the index of the current *future* in the collection.

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.

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*.

Header `hpx/async_combinators/when_some.hpp`

namespace hpx

Functions

template<typename **InputIter**, typename **Container** = vector<future<**typename** *std*::iterator_traits<*InputIter*>::value_typ
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.

template<typename **InputIter**, typename **Container** = vector<future<**typename** *std*::iterator_traits<*InputIter*>::value_ty
future<when_some_result<*Container*>> **when_some_n** (*std*::size_t *n*, Iterator *first*, *std*::size_t *count*, *error_code* &*ec* = *throws*)

The function *when_some_n* is an operator allowing to join on the result of all given futures. It AND-composes all future objects given and returns a new future object representing the same list of futures after n of them finished executing.

Note The future returned by the function *when_some_n* becomes ready when at least *n* argument futures have become ready.

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_n* where count == 0, returns a future with the same elements as the arguments that is immediately ready. Possibly none of the futures in that container are ready. Each future and shared_future is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection. The future returned by *when_some_n* will not throw an exception, but the futures held in the output collection may.

Parameters

- `n`: [in] The number of futures out of the arguments which have to become ready in order for the returned future to get ready.
- `first`: [in] The iterator pointing to the first element of a sequence of *future* or *shared_future* objects for which *when_all* should wait.
- `count`: [in] The number of elements in the sequence starting at *first*.
- `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 **Sequence**>
struct when_some_result

#include <*when_some.hpp*> Result type for *when_some*, contains a sequence of futures and indices pointing to ready futures.

Public Members

std::vector<*std*::size_t> **indices**

List of indices of futures which became ready.

Sequence **futures**

The sequence of futures as passed to hpx::when_some.

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.

Header `hpx/async_cuda/cublas_executor.hpp`

namespace hpx

namespace cuda

namespace experimental

Functions

cublasStatus_t **check_cublas_error**(cublasStatus_t *err*)

struct cublas_exception : **public** *exception*
#include <cublas_executor.hpp>

Public Functions

cublas_exception(**const** *std*::string &*msg*, cublasStatus_t *err*)

cublasStatus_t **get_cublas_errorcode**()

Protected Attributes

cublasStatus_t **err_**

struct cublas_executor : **public** *hpx*::*cuda*::*experimental*::*cuda_executor*
#include <cublas_executor.hpp>

Public Types

using handle_ptr = *std*::shared_ptr<**struct** cublasContext>

Public Functions

cublas_executor(*std*::size_t *device*, cublasPointerMode_t *pointer_mode* = CUBLAS_POINTER_MODE_HOST, bool *event_mode* = false)

~cublas_executor()

template<typename **F**, typename ...**Ts**>
decltype(auto) **post**(*F* &&*f*, *Ts*&&... *ts*)

template<typename **F**, typename ...**Ts**>
decltype(auto) **async_execute**(*F* &&*f*, *Ts*&&... *ts*)

Protected Functions

template<typename **R**, typename ...**Params**, typename ...**Args**>
std::enable_if<*std*::is_same<cublasStatus_t, *R*>::value, *R*>::type **apply** (*R* (**cublas_function*)) *Params*...
, *Args*&&... *args*

template<typename **R**, typename ...**Params**, typename ...**Args**>
std::enable_if<*std*::is_same<cudaError_t, *R*>::value, void>::type **apply** (*R* (**cuda_function*)) *Params*...
, *Args*&&... *args*

template<typename **R**, typename ...**Params**, typename ...**Args**>
hpx::future<**typename** *std*::enable_if<*std*::is_same<cublasStatus_t, *R*>::value, void>::type> **async** (*R* (**cublas_fu*
, *Args*&&... *args*

template<typename **R**, typename ...**Params**, typename ...**Args**>
hpx::future<**typename** *std*::enable_if<*std*::is_same<cudaError_t, *R*>::value, void>::type> **async** (*R* (**cuda_functi*
, *Args*&&... *args*

cublasHandle_t **get_handle** ()

Protected Attributes

handle_ptr **handle_**

cublasPointerMode_t **pointer_mode_**

Header `hpx/async_cuda/cuda_event.hpp`

namespace hpx

namespace cuda

namespace experimental

struct cuda_event_pool
#include <cuda_event.hpp>

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_**

Header hpx/async_cuda/cuda_exception.hpp

namespace hpx

namespace cuda

namespace experimental

Functions

cudaError_t **check_cuda_error**(cudaError_t *err*)

struct cuda_exception : **public** *exception*
#include <cuda_exception.hpp>

Public Functions

cuda_exception(**const** *std*::string &*msg*, cudaError_t *err*)

cudaError_t **get_cuda_errorcode**()

Protected Attributes

cudaError_t **err_**

Header `hpx/async_cuda/cuda_executor.hpp`

namespace hpx

namespace cuda

namespace experimental

struct cuda_executor : **public** *hpx::cuda::experimental::cuda_executor_base*

#include <cuda_executor.hpp> Subclassed by *hpx::cuda::experimental::cublas_executor*

Public Functions

cuda_executor (*std*::size_t *device*, bool *event_mode* = true)

~cuda_executor ()

template<typename **F**, typename ...**Ts**>
decltype(auto) **post** (*F &&f*, *Ts&&... ts*)

template<typename **F**, typename ...**Ts**>
decltype(auto) **async_execute** (*F &&f*, *Ts&&... ts*)

Protected Functions

template<typename **R**, typename ...**Params**, typename ...**Args**>
void **apply** (*R* (**cuda_function*)) *Params...*
, *Args&&... args*

template<typename **R**, typename ...**Params**, typename ...**Args**>
hpx::future<void> **async** (*R* (**cuda_kernel*)) *Params...*
, *Args&&... args*

struct cuda_executor_base

#include <cuda_executor.hpp> Subclassed by *hpx::cuda::experimental::cuda_executor*

Public Types

using future_type = *hpx*::future<void>

Public Functions

cuda_executor_base (*std*::size_t *device*, bool *event_mode*)

future_type **get_future** ()

Protected Attributes

int **device_**

bool **event_mode_**

cudaStream_t **stream_**

std::shared_ptr<*hpx*::*cuda*::*experimental*::*target*> **target_**

Header `hpx/async_cuda/cuda_future.hpp`

`namespace hpx`

`namespace cuda`

`namespace experimental`

Typedefs

`using print_on` = debug::enable_print<false>

`using event_mode` = *std*::true_type

`using callback_mode` = *std*::false_type

Functions

`static constexpr print_on hpx::cuda::experimental::cud_debug("CUDAFUT")`

`struct enable_user_polling`

#include <cuda_future.hpp>

Public Functions

`enable_user_polling`(*std*::string **const** &*pool_name* = "")

`~enable_user_polling`()

Private Members

std::string **pool_name_**

Header hpx/async_cuda/get_targets.hpp

namespace hpx

namespace cuda

namespace experimental

Functions

std::vector<*target*> **get_local_targets**()

void **print_local_targets**()

Header hpx/async_cuda/target.hpp

namespace hpx

namespace cuda

namespace experimental

Functions

target &**get_default_target**()

struct target

#include <target.hpp>

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**()

native_handle_type **const** &**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
#include <target.hpp>

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

async_distributed

The contents of this module can be included with the header `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 `hpx/modules/async_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.

Header `hpx/async.hpp`

Header `hpx/async_distributed/applier/applier.hpp`

Header `hpx/async_distributed/applier/apply.hpp`

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 **ap**

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 **ap**

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*)

Header `hpx/async_distributed/applier/apply_callback.hpp`

namespace hpx

Functions

template<typename **Action**, typename **Callback**, typename ...**Ts**>
bool **apply_p_cb** (*naming*::id_type **const** &*gid*, *threads*::*thread_priority priority*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Action**, typename **Callback**, typename ...**Ts**>
bool **apply_cb** (*naming*::id_type **const** &*gid*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Component**, typename **Signature**, typename **Derived**, typename **Callback**, typename ...**Ts**>
bool **apply_cb** (*hpx*::*actions*::*basic_action*<*Component*, *Signature*, *Derived*>, *naming*::id_type **const** &*gid*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Action**, typename **DistPolicy**, typename **Callback**, typename ...**Ts**>
std::enable_if<*traits*::is_distribution_policy<*DistPolicy*>::value, bool>::type **apply_p_cb** (*DistPolicy* **const** &*policy*, *threads*::*thread_priority priority*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Action**, typename **DistPolicy**, typename **Callback**, typename ...**Ts**>
std::enable_if<*traits*::is_distribution_policy<*DistPolicy*>::value, bool>::type **apply_cb** (*DistPolicy* **const** &*policy*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Component**, typename **Signature**, typename **Derived**, typename **DistPolicy**, typename **Callbac**
std::enable_if<*traits*::is_distribution_policy<*DistPolicy*>::value, bool>::type **apply_cb**(*hpx*::*actions*::*basic_action*<*Componen*
Signature, *Derived*>, *DistPolicy* **const** &*policy*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
bool **apply_p_cb**(*Continuation* &&*c*, *naming*::address &&*addr*, *naming*::id_type **const** &*gid*, *threads*::*thread_priority priority*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
bool **apply_p_cb**(*Continuation* &&*c*, *naming*::id_type **const** &*gid*, *threads*::*thread_priority priority*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Action**, typename **Continuation**, typename **Callback**, typename ...**Ts**>
bool **apply_cb**(*Continuation* &&*c*, *naming*::id_type **const** &*gid*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Component**, typename **Continuation**, typename **Signature**, typename **Derived**, typename **Callb**
bool **apply_cb**(*Continuation* &&*c*, *hpx*::*actions*::*basic_action*<*Component*, *Signature*, *Derived*>, *naming*::id_type **const** &*gid*, *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 **ap**

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 **ap**

template<typename **Component**, typename **Continuation**, typename **Signature**, typename **Derived**, typename **DistP**
std::enable_if<*traits*::is_distribution_policy<*DistPolicy*>::value, bool>::type **apply_cb** (*Continuation* &&*c*, *hpx*::*actions*::*basic_action*<*Componen*... *Signature*, *Derived*>, *DistPolicy* **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_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** *util*::decay<*Callback*>::type, **typename** *util*::decay<*Ts*>::type...> **apply_c_p_**

template<typename **Action**, typename **Callback**, typename ...**Ts**>
struct apply_c_p_cb_impl
#include <apply_callback.hpp>

Public Types

typedef *util*::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<*std*::size_t... **Is**>
void **apply_action**(*util*::*index_pack*<*Is*...>)

Private Members

naming::id_type **contid_**

naming::address **addr_**

naming::id_type **id_**

threads::*thread_priority* **p_**

Callback **cb_**

tuple_type **args_**

Header `hpx/async_distributed/applier/apply_continue.hpp`

namespace 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*)

Header `hpx/async_distributed/applier/apply_continue_callback.hpp`

namespace hpx

Functions

template<typename **Action**, typename **Cont**, typename **Callback**, typename ...**Ts**>
bool **apply_continue_cb**(*Cont* &&*cont*, *naming*::id_type **const** &*gid*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Component**, typename **Signature**, typename **Derived**, typename **Cont**, typename **Callback**, typename ...**Ts**>
bool **apply_continue_cb**(*hpx*::*actions*::*basic_action*<*Component*, *Signature*, *Derived*>, *Cont* &&*cont*, *naming*::id_type **const** &*gid*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Action**, typename **Callback**, typename ...**Ts**>
bool **apply_continue_cb**(*naming*::id_type **const** &*cont*, *naming*::id_type **const** &*gid*, *Callback* &&*cb*, *Ts*&&... *vs*)

template<typename **Component**, typename **Signature**, typename **Derived**, typename **Callback**, typename ...**Ts**>
bool **apply_continue_cb**(*hpx*::*actions*::*basic_action*<*Component*, *Signature*, *Derived*>, *naming*::id_type **const** &*cont*, *naming*::id_type **const** &*gid*, *Callback* &&*cb*, *Ts*&&... *vs*)

Header `hpx/async_distributed/applier/apply_continue_fwd.hpp`

Header `hpx/async_distributed/applier/apply_helper.hpp`

namespace hpx

Functions

bool **is_pre_startup**()

Header `hpx/async_distributed/applier/bind_naming_wrappers.hpp`

namespace hpx

namespace applier

Functions

bool **bind_gid_local**(*naming*::gid_type **const**&, *naming*::address **const**&, *error_code* &*ec* = *throws*)

void **unbind_gid_local**(*naming*::gid_type **const**&, *error_code* &*ec* = *throws*)

bool **bind_range_local**(*naming*::gid_type **const**&, *std*::size_t, *naming*::address **const**&, *std*::size_t, *error_code* &*ec* = *throws*)

void **unbind_range_local**(*naming*::gid_type **const**&, *std*::size_t, *error_code* &*ec* = *throws*)

Header `hpx/async_distributed/applier/register_apply_colocated.hpp`

Defines

HPX_REGISTER_APPLY_COLOCATED_DECLARATION (*Action*, *Name*)

HPX_REGISTER_APPLY_COLOCATED (*action*, *name*)

Header `hpx/async_distributed/applier/trigger.hpp`

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*)

Header `hpx/async_distributed/applier_fwd.hpp`

namespace hpx

namespace applier

Functions

applier &**get_applier** ()

The function *get_applier* returns a reference to the (thread specific) applier instance.

applier ***get_applier_ptr** ()

The function *get_applier* returns a pointer to the (thread specific) applier instance. The returned pointer is NULL if the current thread is not known to HPX or if the runtime system is not active.

namespace applier

The namespace *applier* contains all definitions needed for the class *hpx::applier::applier* and its related functionality. This namespace is part of the HPX core module.

Header hpx/async_distributed/apply.hpp

Header hpx/async_distributed/async.hpp

namespace hpx

Functions

template<typename **Action**, typename **F**, typename ...**Ts**>
auto **async** (*F &&f*, *Ts&&... ts*)

Header hpx/async_distributed/async_callback.hpp

namespace hpx

Functions

template<typename **Action**, typename **F**, typename ...**Ts**>
auto **async_cb** (*F &&f*, *Ts&&... ts*)

template<typename **F**, typename ...**Ts**>
auto **async_cb** (*F &&f*, *Ts&&... ts*)

Header hpx/async_distributed/async_callback_fwd.hpp

Header hpx/async_distributed/async_continue.hpp

namespace hpx

Functions

template<typename **Action**, typename **Cont**, typename ...**Ts**>
lcos::future<**typename** *traits*::promise_local_result<**typename** detail::result_of_async_continue<*Action*, *Cont*>::type>::type>

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

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<**typenam**

template<typename **Component**, typename **Signature**, typename **Derived**, typename **Cont**, typename **DistPolicy**, typ
std::enable_if<*traits*::is_distribution_policy<*DistPolicy*>::value, *lcos*::future<**typename** *traits*::promise_local_result<**typenam**

Header `hpx/async_distributed/async_continue_callback.hpp`

`namespace hpx`

Functions

template<typename **`Action`**, typename **`Cont`**, typename **`Callback`**, typename ...**`Ts`**>
lcos::future<**`typename`** *traits*::promise_local_result<**`typename`** detail::result_of_async_continue<*Action*, *Cont*>::type>::type>

template<typename **`Component`**, typename **`Signature`**, typename **`Derived`**, typename **`Cont`**, typename **`Callback`**, typena
lcos::future<**`typename`** *traits*::promise_local_result<**`typename`** detail::result_of_async_continue<*Derived*, *Cont*>::type>::type

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<**`typenam`**

template<typename **`Component`**, typename **`Signature`**, typename **`Derived`**, typename **`Cont`**, typename **`DistPolicy`**, typ

std::enable_if<*traits*::is_distribution_policy<*DistPolicy*>::value, *lcos*::future<**typename** *traits*::promise_local_result<**typenam**

Header `hpx/async_distributed/async_continue_callback_fwd.hpp`

Header `hpx/async_distributed/async_continue_fwd.hpp`

Header `hpx/async_distributed/dataflow.hpp`

namespace hpx

Functions

template<typename **Action**, typename **T0**, typename ...**Ts**, typename **Enable** = **typename** *std*::enable_if<*traits*::is_action<*A*
auto **dataflow** (*T0 &&t0*, *Ts&&... ts*)

template<typename **Action**, typename **Allocator**, typename **T0**, typename ...**Ts**, typename **Enable** = **typename** *std*::ena
auto **dataflow_alloc** (*Allocator* **const** *&alloc*, *T0 &&t0*, *Ts&&... ts*)

Header `hpx/async_distributed/sync.hpp`

namespace hpx

Functions

template<typename **Action**, typename **F**, typename ...**Ts**>
auto **sync** (*F &&f*, *Ts&&... ts*)

Header `hpx/modules/async_distributed.hpp`

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.

Header `hpx/async_local/apply.hpp`

Header `hpx/async_local/async.hpp`

Header `hpx/async_local/async_fwd.hpp`

namespace hpx

Functions

template<typename **Action**, typename **F**, typename ...**Ts**>
auto **async**(*F* &&*f*, *Ts*&&... *ts*)

Header `hpx/async_local/dataflow.hpp`

Header `hpx/async_local/sync.hpp`

Header `hpx/async_local/sync_fwd.hpp`

namespace hpx

Functions

template<typename **Action**, typename **F**, typename ...**Ts**>
auto **sync**(*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.

Header `hpx/async_mpi/mpi_executor.hpp`

namespace hpx

namespace mpi

namespace experimental

struct executor
#include <mpi_executor.hpp>

Public Types

using execution_category = *parallel::execution::parallel_execution_tag*

using executor_parameters_type = *hpx::parallel::execution::static_chunk_size*

Public Functions

constexpr executor(MPI_Comm *communicator* = MPI_COMM_WORLD)

template<typename **F**, typename ...**Ts**>
decltype(auto) **async_execute**(*F* &&*f*, *Ts*&&... *ts*) **const**

std::size_t **in_flight_estimate**() **const**

Private Members

MPI_Comm **communicator_**

Header `hpx/async_mpi/mpi_future.hpp`

namespace hpx

namespace mpi

namespace experimental

Typedefs

using print_on = *debug*::enable_print<false>

Functions

static constexpr print_on hpx::mpi::experimental::mpi_debug("MPI_FUT")

void **set_error_handler**()

hpx::future<void> **get_future**(MPI_Request *request*)

void **poll**()

void **wait**()

template<typename **F**>
void **wait**(*F* &&*f*)

void **init**(bool *init_mpi* = false, *std*::string **const** &*pool_name* = "", bool *init_errorhandler* = false)

void **finalize**(*std*::string **const** &*pool_name* = "")

template<typename ...**Args**>
void **debug**(*Args*&&... *args*)

struct enable_user_polling
#include <mpi_future.hpp>

Public Functions

enable_user_polling(*std*::string **const** &*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.

Header `hpx/batch_environments/alps_environment.hpp`

namespace hpx

namespace util

namespace batch_environments

struct alps_environment

#include <alps_environment.hpp>

Public Functions

alps_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 Members

std::size_t **node_num_**

std::size_t **num_threads_**

std::size_t **num_localities_**

bool **valid_**

Header `hpx/batch_environments/batch_environment.hpp`

namespace hpx

namespace util

struct batch_environment

#include <batch_environment.hpp>

Public Types

typedef *std*::map<*boost*::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_**

node_map_type **nodes_**

std::size_t **num_localities_**

std::string **batch_name_**

bool **debug_**

Header `hpx/batch_environments/pbs_environment.hpp`

namespace hpx

namespace util

namespace batch_environments

struct pbs_environment
#include <pbs_environment.hpp>

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_**

Header `hpx/batch_environments/slurm_environment.hpp`

namespace hpx

namespace util

namespace batch_environments

struct slurm_environment
#include <slurm_environment.hpp>

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.

Header `hpx/cache/entries/entry.hpp`

namespace hpx

namespace util

namespace cache

namespace entries

template<typename **Value**, typename **Derived**>
class entry : **private** *hpx*::*util*::*cache*::*entries*::detail::less_than_comparable<detail::derived<*Value*, *Derive*

#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

entry()

Any cache entry has to be default constructible.

entry(*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, other wise 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.

Header `hpx/cache/entries/fifo_entry.hpp`

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*.

std::chrono::steady_clock::time_point **const** &**get_creation_time** () **const**

Private Types

typedef entry<Value, fifo_entry<Value>> **base_type**

Private Members

std::chrono::steady_clock::time_point **insertion_time_**

Friends

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).

Header `hpx/cache/entries/lfu_entry.hpp`

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

bool **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).

Header `hpx/cache/entries/lru_entry.hpp`

namespace hpx

namespace util

namespace cache

namespace entries

template<typename **Value**>
class lru_entry : **public** *hpx*::*util*::*cache*::*entries*::entry<*Value*, *lru_entry*<*Value*>>

#include <hpx/cache/entries/lru_entry.hpp> The lru_entry type can be used to store arbitrary values in a cache. Using this type as the cache's entry type makes sure that the least recently used entries are discarded from the cache first.

Note The lru_entry conforms to the CacheEntry concept.

Note This type can be used to model a 'most recently used' cache policy if it is used with a std::greater as the caches' UpdatePolicy (instead of the default std::less).

Template Parameters

- `Value`: The data type to be stored in a cache. It has to be default constructible, copy constructible and less_than_comparable.

Public Functions

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).

Header `hpx/cache/entries/size_entry.hpp`

namespace hpx

namespace util

namespace cache

namespace entries

template<typename **Value**, typename **Derived**>

class size_entry : **public** *hpx*::*util*::*cache*::*entries*::entry<*Value*, detail::size_derived<*Value*, *Derived*>::ty

#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.

Header `hpx/cache/local_cache.hpp`

namespace hpx

namespace util

namespace cache

template<typename **Key**, typename **Entry**, typename **UpdatePolicy** = *std*::less<*Entry*>, typename **InsertPoli**

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

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

local_cache(*size_type max_size* = 0, *update_policy_type* **const** &*up* = *update_policy_type*(), *insert_policy_type* **const** &*ip* = *insert_policy_type*())

Construct an instance of a *local_cache*.

Parameters

- `max_size`: [in] The maximal size this cache is allowed to reach any time. The default is zero (no size limitation). The unit of this value is usually determined by the unit of the values returned by the entry's *get_size* function.
- `up`: [in] An instance of the *UpdatePolicy* to use for this cache. The default is to use a default constructed instance of the type as defined by the *UpdatePolicy* template parameter.
- `ip`: [in] An instance of the *InsertPolicy* to use for this cache. The default is to use a default constructed instance of the type as defined by the *InsertPolicy* template parameter.

local_cache(*local_cache* &&*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.

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 *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 value.

bool **insert**(*key_type* **const** &*k*, *value_type* **const** &*val*)

Insert a new element into this cache.

Note This function invokes both, the insert policy as provided to the constructor and the function *entry::insert* of the newly constructed entry instance. If either of these functions returns false the key/value pair doesn't get inserted into the cache and the *insert* function will return *false*. Other reasons for this function to fail (return *false*) are a) the key/value pair is already held in the cache or b) inserting the new value into the cache maxed out its capacity and it was not possible to free any of the existing entries.

Return This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

Parameters

- `k`: [in] The key for the entry which should be added to the cache.
- `value`: [in] The value which should be added to the cache.

bool **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 *entry::insert* of the provided entry instance. If either of these functions returns false the key/value pair doesn't get inserted into the cache and the *insert* function will return *false*. Other reasons for this function to fail (return *false*) are a) the key/value pair is already held in the cache or b) inserting the new value into the cache maxed out its capacity and it was not possible to free any of the existing entries.

Return This function returns *true* if the entry has been successfully added to the cache, otherwise it returns *false*.

Parameters

- `k`: [in] The key for the entry which should be added to the cache.
- `value`: [in] The entry which should be added to the cache.

bool **update**(*key_type* **const** &*k*, *value_type* **const** &*val*)

Update an existing element in this cache.

Note The function will call the entry's *entry::touch* function if the indexed value is found in the cache.

Note The difference to the other overload of the *insert* function is that this overload replaces the cached value only, while the other overload replaces the whole cache entry, updating the cache entry properties.

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.

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 retplaces 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).

void **clear**()
Clear the cache.

Unconditionally removes all stored entries from the cache.

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

statistics_type &**get_statistics**()

Protected Functions

bool **free_space**(long *num_free*)

Private Types

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

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_**

Header `hpx/cache/lru_cache.hpp`

namespace hpx

namespace util

namespace cache

template<typename **Key**, typename **Entry**, typename **Statistics** = *statistics*::*no_statistics*>
class lru_cache

#include <hpx/cache/lru_cache.hpp> The lru_cache implements the basic functionality needed for a local (non-distributed) LRU cache.

Template Parameters

- `Key`: The type of the keys to use to identify the entries stored in the cache
- `Entry`: The type of the items to be held in the cache.
- `Statistics`: A (optional) type allowing to collect some basic statistics about the operation of the cache instance. The type must conform to the CacheStatistics concept. The default value is the type statistics::no_statistics which does not collect any numbers, but provides empty stubs allowing the code to compile.

Public Types

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.
- `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.

template<typename **Func**>
size_type **erase**(*Func* **const** &*ep*)

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*.

size_type **erase**()

Remove all stored entries from the cache.

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).

size_type **clear**()

Clear the cache.

Unconditionally removes all stored entries from the cache.

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

statistics_type &**get_statistics**()

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_**

Header `hpx/cache/policies/always.hpp`

namespace hpx

namespace util

namespace cache

namespace policies

template<typename **Entry**>
struct always
#include <always.hpp>

Public Functions

bool **operator()** (Entry **const**&)

Header hpx/cache/statistics/local_full_statistics.hpp

namespace hpx

namespace util

namespace cache

namespace statistics

class local_full_statistics : **public** *hpx*::*util*::*cache*::*statistics*::*local_statistics*
#include <local_full_statistics.hpp>

Public Functions

std::int64_t **get_get_entry_count** (bool *reset*)
The function *get_get_entry_count* returns the number of invocations of the get_entry() API function of the cache.

std::int64_t **get_insert_entry_count** (bool *reset*)
The function *get_insert_entry_count* returns the number of invocations of the insert_entry() API function of the cache.

std::int64_t **get_update_entry_count** (bool *reset*)
The function *get_update_entry_count* returns the number of invocations of the update_entry() API function of the cache.

std::int64_t **get_erase_entry_count** (bool *reset*)
The function *get_erase_entry_count* returns the number of invocations of the erase() API function of the cache.

std::int64_t **get_get_entry_time** (bool *reset*)
The function *get_get_entry_time* returns the overall time spent executing of the get_entry() API function of the cache.

std::int64_t **get_insert_entry_time** (bool *reset*)
The function *get_insert_entry_time* returns the overall time spent executing of the insert_entry() API function of the cache.

std::int64_t **get_update_entry_time**(bool *reset*)
The function *get_update_entry_time* returns the overall time spent executing of the update_entry() API function of the cache.

std::int64_t **get_erase_entry_time**(bool *reset*)
The function *get_erase_entry_time* returns the overall time spent executing of the erase() API function of the cache.

Private Functions

std::int64_t **get_and_reset_value**(*std*::int64_t &*value*, bool *reset*)

Private Members

api_counter_data **get_entry_**

api_counter_data **insert_entry_**

api_counter_data **update_entry_**

api_counter_data **erase_entry_**

Friends

friend hpx::util::cache::statistics::update_on_exit

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*)

static *std*::uint64_t **now**()

Header `hpx/cache/statistics/local_statistics.hpp`

namespace hpx

namespace util

namespace cache

namespace statistics

class local_statistics : **public** *hpx*::*util*::*cache*::*statistics*::*no_statistics*
#include <local_statistics.hpp> Subclassed by *hpx::util::cache::statistics::local_full_statistics*

Public Functions

local_statistics()

std::size_t **get_and_reset**(*std*::size_t &*value*, bool *reset*)

std::size_t **hits**() **const**

std::size_t **misses**() **const**

std::size_t **insertions**() **const**

std::size_t **evictions**() **const**

std::size_t **hits**(bool *reset*)

std::size_t **misses**(bool *reset*)

std::size_t **insertions**(bool *reset*)

std::size_t **evictions**(bool *reset*)

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.

Private Members

std::size_t **hits_**

std::size_t **misses_**

std::size_t **insertions_**

std::size_t **evictions_**

Header `hpx/cache/statistics/no_statistics.hpp`

namespace hpx

namespace util

namespace cache

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
#include <*no_statistics.hpp*> 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.

std::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.

std::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.

std::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.

std::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.

std::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.

std::int64_t **get_insert_entry_time**(bool)
The function *get_insert_entry_time* returns the overall time spent executing of the insert_entry() API function of the cache.

std::int64_t **get_update_entry_time**(bool)
The function *get_update_entry_time* returns the overall time spent executing of the update_entry() API function of the cache.

std::int64_t **get_erase_entry_time**(bool)
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*)

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.

Header `hpx/checkpoint/checkpoint.hpp`

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

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.

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.

template<typename **T**, typename ...**Ts**, typename **U** = **typename** *std*::enable_if<!*hpx*::*traits*::is_launch_policy<*T*>::value &

hpx::future<*checkpoint*> **save_checkpoint** (*T* &&*t*, *Ts*&&... *ts*)

Save_checkpoint

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- `T`: Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- `Ts`: More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- `U`: This parameter is used to make sure that T is not a launch policy or a checkpoint. This forces the compiler to choose the correct overload.

Parameters

- `t`: A container to restore.
- `ts`: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

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.

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.

template<typename **T**, typename ...**Ts**, typename **U** = **typename** *std*::enable_if<!*std*::is_same<**typename** *std*::decay<*T*>
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.

template<typename **T**, typename ...**Ts**>
checkpoint **save_checkpoint**(*hpx*::*launch*::sync_policy *sync_p*, *checkpoint* &&*c*, *T* &&*t*, *Ts*&&... *ts*)

Save_checkpoint - Sync_policy overload & pre-init. checkpoint

Save_checkpoint takes any number of objects which a user may wish to store and returns a future to a checkpoint object. This function can also store a component either by passing a shared_ptr to the component or by passing a component's client instance to save_checkpoint. Additionally the function can take a policy as a first object which changes its behavior depending on the policy passed to it. Most notably, if a sync policy is used save_checkpoint will simply return a checkpoint object.

Template Parameters

- `T`: Containers passed to save_checkpoint to be serialized and placed into a checkpoint object.
- `Ts`: More containers passed to save_checkpoint to be serialized and placed into a checkpoint object.

Parameters

- `sync_p`: hpx::launch::sync_policy
- `c`: Takes a pre-initialized checkpoint to copy data into.
- `t`: A container to restore.
- `ts`: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

Return Save_checkpoint which is passed hpx::launch::sync_policy will return a checkpoint which contains the serialized values checkpoint.

template<typename **T**, typename ...**Ts**, typename **U** = **typename** *std*::enable_if<!*hpx*::*traits*::is_launch_policy<*T*>::value &
hpx::future<*checkpoint*> **prepare_checkpoint**(*T* **const** &*t*, *Ts* **const**&... *ts*)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

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.

template<typename **T**, typename ...**Ts**>
hpx::future<*checkpoint*> **prepare_checkpoint**(*checkpoint* &&*c*, *T* **const** &*t*, *Ts* **const**&... *ts*)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

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

- `c`: Takes a pre-initialized checkpoint to prepare
- `t`: A container to restore.
- `ts`: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

template<typename **T**, typename ...**Ts**, typename **U** = **typename** *std*::enable_if<!*std*::is_same<*T*, *checkpoint*>::value>::typ
hpx::future<*checkpoint*> **prepare_checkpoint** (*hpx*::*launch p*, *T* **const** &*t*, *Ts* **const**&... *ts*)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

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

- `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.

template<typename **T**, typename ...**Ts**>
hpx::future<*checkpoint*> **prepare_checkpoint** (*hpx*::*launch p*, *checkpoint* &&*c*, *T* **const** &*t*, *Ts* **const**&... *ts*)

prepare_checkpoint

prepare_checkpoint takes the containers which have to be filled from the byte stream by a subsequent restore_checkpoint invocation. prepare_checkpoint will calculate the necessary buffer size and will return an appropriately sized checkpoint object.

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

- `p`: Takes an HPX launch policy. Allows the user to change the way the function is launched i.e. async, sync, etc.
- `c`: Takes a pre-initialized checkpoint to prepare
- `t`: A container to restore.
- `ts`: Other containers to restore Containers must be in the same order that they were inserted into the checkpoint.

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.

class checkpoint

#include <checkpoint.hpp> Checkpoint Object

Checkpoint is the container object which is produced by save_checkpoint and is consumed by a restore_checkpoint. A checkpoint may be moved into the save_checkpoint object to write the byte stream to the pre-created checkpoint object.

Checkpoints are able to store all containers which are able to be serialized including components.

Public Types

using const_iterator = *std*::vector::const_iterator

Public Functions

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

template<typename **Archive**>
void **serialize** (*Archive &arch*, **const** unsigned int *version*)

Private Members

std::vector<char> **data_**

Friends

friend hpx::util::hpx::serialization::access

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.

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.

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.

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.

Header `hpx/checkpoint_base/checkpoint_data.hpp`

namespace hpx

namespace util

Functions

template<typename **Container**, typename ...**Ts**>
void **save_checkpoint_data**(*Container &data*, *Ts&&... ts*)

save_checkpoint_data

Save_checkpoint_data takes any number of objects which a user may wish to store in the given container.

Template Parameters
- `Container`: Container used to store the check-pointed data.
- `Ts`: Types of variables to checkpoint

Parameters
- `cont`: Container instance used to store the checkpoint data
- `ts`: Variable instances to be inserted into the checkpoint.

template<typename ...**Ts**>
std::size_t **prepare_checkpoint_data**(*Ts* **const**&... *ts*)

prepare_checkpoint_data

prepare_checkpoint_data takes any number of objects which a user may wish to store in a subsequent save_checkpoint_data operation. The function will return the number of bytes necessary to store the data that will be produced.

Template Parameters
- `Ts`: Types of variables to checkpoint

Parameters
- `ts`: Variable instances to be inserted into the checkpoint.

template<typename **Container**, typename ...**Ts**>

void **restore_checkpoint_data**(*Container* **const** &*cont*, *Ts*&... *ts*)
restore_checkpoint_data

restore_checkpoint_data takes any number of objects which a user may wish to restore from the given container. The sequence of objects has to correspond to the sequence of objects for the corresponding call to save_checkpoint_data that had used the given container instance.

Template Parameters

- `Container`: Container used to restore the check-pointed data.
- `Ts`: Types of variables to restore

Parameters

- `cont`: Container instance used to restore the checkpoint data
- `ts`: Variable instances to be restored from the container

collectives

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.

Header `hpx/collectives/all_gather.hpp`

namespace hpx

namespace lcos

Functions

template<typename **T**>
hpx::future<*std*::vector<*T*>> **all_gather**(char **const** **basename*, *hpx*::future<*T*> &&*result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)
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.

Note Each all_gather operation has to be accompanied with a unique usage of the *HPX_REGISTER_ALLTOALL* macro to define the necessary internal facilities used by *all_gather*.

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`: A future referring to 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. root_site The site that is

responsible for creating the all_gather support object. This value is optional and defaults to '0' (zero).

template<typename **T**>
hpx::future<*std*::vector<**typename** *std*::decay<*T*>::type>> **all_gather**(char **const** **basename*, *T* &&*result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

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.

Note Each all_gather operation has to be accompanied with a unique usage of the *HPX_REGISTER_ALLTOALL* macro to define the necessary internal facilities used by *all_gather*.

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. root_site The site that is responsible for creating the all_gather support object. This value is optional and defaults to '0' (zero).

Header `hpx/collectives/all_reduce.hpp`

namespace hpx

namespace lcos

Functions

template<typename **T**, typename **F**>
hpx::future<*T*> **all_reduce**(char **const** **basename*, *hpx*::future<*T*> *result*, *F* &&*op*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

AllReduce a set of values from different call sites

This function receives a set of values that are the result of applying a given operator on values supplied from all call sites operating on the given base name.

Note Each all_reduce operation has to be accompanied with a unique usage of the *HPX_REGISTER_ALLREDUCE* macro to define the necessary internal facilities used by *all_reduce*.

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

- `basename`: The base name identifying the all_reduce operation
- `local_result`: A future referring to 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. root_site The site that is responsible for creating the all_reduce support object. This value is optional and defaults to '0' (zero).

template<typename **T**, typename **F**>
hpx::future<*std*::decay_t<*T*>> **all_reduce**(char **const** **basename*, *T* &&*result*, *F* &&*op*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

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.

Note Each all_reduce operation has to be accompanied with a unique usage of the *HPX_REGISTER_ALLREDUCE* macro to define the necessary internal facilities used by *all_reduce*.

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. root_site The site that is responsible for creating the all_reduce support object. This value is optional and defaults to '0' (zero).

Header `hpx/collectives/all_to_all.hpp`

namespace hpx

namespace lcos

Functions

template<typename **T**>
hpx::future<*std*::vector<*T*>> **all_to_all** (char **const** **basename*, *hpx*::future<*std*::vector<*T*>> &&*result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

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.

Note Each all_to_all operation has to be accompanied with a unique usage of the *HPX_REGISTER_ALLTOALL* macro to define the necessary internal facilities used by *all_to_all*.

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`: A future referring to 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 whatever hpx::get_locality_id() returns. root_site The site that is responsible for creating the all_to_all support object. This value is optional and defaults to '0' (zero).

template<typename **T**>
hpx::future<*std*::vector<**typename** *std*::decay<*T*>::type>> **all_to_all** (char **const** **basename*, *T* &&*result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

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.

Note Each all_to_all operation has to be accompanied with a unique usage of the *HPX_REGISTER_ALLTOALL* macro to define the necessary internal facilities used by *all_to_all*.

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 whatever hpx::get_locality_id() returns. root_site The site that is

responsible for creating the all_to_all support object. This value is optional and defaults to '0' (zero).

Header hpx/collectives/barrier.hpp

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 localities.

Header `hpx/collectives/broadcast.hpp`

Header `hpx/collectives/broadcast_direct.hpp`

namespace hpx

namespace lcos

Functions

template<typename Action, typename ArgN, ...>hpx::future<std::vector<decltype(Actic

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.

template<typename Action, typename ArgN, ...>void hpx::lcos::broadcast_apply(std::v

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.

template<typename Action, typename ArgN, ...>hpx::future< std::vector<decltype(Acti

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.

template<typename Action, typename ArgN, ...>void hpx::lcos::broadcast_apply_with_i

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.
- `argN`: [in] Any number of arbitrary arguments (passed by const reference) which will be forwarded to the action invocation.

Header `hpx/collectives/fold.hpp`

namespace hpx

namespace lcos

Functions

`template<typename Action, typename FoldOp, typename Init, typename ArgN, ...>hpx::f`

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::f`

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::f`

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::f`

Perform a distributed inverse folding operation.

The function *hpx::lcos::inverse_fold_with_index* performs an inverse distributed folding operation over results returned from action invocations on a given set of global identifiers. The action can be either plain action (in which case the global identifiers have to refer to localities) or a component action (in which case the global identifiers have to refer to instances of a component type which exposes the action.

The function passes the index of the global identifier in the given list of identifiers as the last argument to the action.

Note The type of the initial value must be convertible to the result type returned from the invoked action.

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.

Header `hpx/collectives/gather.hpp`

`namespace hpx`

`namespace lcos`

Functions

template<typename **T**>
hpx::future<*std*::vector<*T*>> **gather_here**(char **const** **basename*, *hpx*::future<*T*> *result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1))

Gather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Note Each gather operation has to be accompanied with a unique usage of the *HPX_REGISTER_GATHER* macro to define the necessary internal facilities used by *gather_here* and *gather_there*

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`: A future referring to 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.

template<typename **T**>
hpx::future<*std*::vector<*T*>> **gather_there**(char **const** **basename*, *hpx*::future<*T*> *result*, *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

Gather a given value at the given call site

This function transmits the value given by *result* to a central gather site (where the corresponding *gather_here* is executed)

Note Each gather operation has to be accompanied with a unique usage of the *HPX_REGISTER_GATHER* macro to define the necessary internal facilities used by *gather_here* and *gather_there*

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`: A future referring to 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.

template<typename **T**>

hpx::future<*std*::vector<**typename** *std*::decay<*T*>::type>> **gather_here**(char **const** **basename*, *T* &&*result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1))

Gather a set of values from different call sites

This function receives a set of values from all call sites operating on the given base name.

Note Each gather operation has to be accompanied with a unique usage of the *HPX_REGISTER_GATHER* macro to define the necessary internal facilities used by *gather_here* and *gather_there*

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.

template<typename **T**>
hpx::future<*std*::vector<**typename** *std*::decay<*T*>::type>> **gather_there**(char **const** **basename*, *T* &&*result*, *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

Gather a given value at the given call site

This function transmits the value given by *result* to a central gather site (where the corresponding *gather_here* is executed)

Note Each gather operation has to be accompanied with a unique usage of the *HPX_REGISTER_GATHER* macro to define the necessary internal facilities used by *gather_here* and *gather_there*

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.

Header `hpx/collectives/latch.hpp`

namespace hpx

namespace lcos

class latch : **public** *components*::client_base<*latch*, *lcos*::*server*::latch>
#include <latch.hpp>

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**

Header `hpx/collectives/reduce.hpp`

namespace hpx

namespace lcos

Functions

template<typename Action, typename ReduceOp, typename ArgN, ...>hpx::future<decltyp

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.

template<typename Action, typename ReduceOp, typename ArgN, ...>hpx::future<decltyp

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.

Header `hpx/collectives/scatter.hpp`

namespace hpx

namespace lcos

Functions

template<typename **T**>
hpx::future<*T*> **scatter_from**(char **const** **basename*, *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1), *std*::size_t *root_site* = 0)

Scatter (receive) a set of values to different call sites

This function receives an element of a set of values operating on the given base name.

Scatter (receive) a set of values to different call sites

Note Each scatter operation has to be accompanied with a unique usage of the *HPX_REGISTER_SCATTER* macro to define the necessary internal facilities used by *scatter_from* and *scatter_to*

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`: A future referring to the value to transmit to the central scatter point from this call site.
- `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.

This function receives an element of a set of values operating on the given base name.

Note Each scatter operation has to be accompanied with a unique usage of the *HPX_REGISTER_SCATTER* macro to define the necessary internal facilities used by *scatter_from* and *scatter_to*

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.
- `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.

template<typename **T**>
hpx::future<*T*> **scatter_to**(char **const** **basename*, *hpx*::future<*std*::vector<*T*>> *result*, *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1))

Scatter (send) a part of the value set at the given call site

This function transmits the value given by *result* to a central scatter site (where the corresponding *scatter_from* is executed)

Note Each scatter operation has to be accompanied with a unique usage of the *HPX_REGISTER_SCATTER* macro to define the necessary internal facilities used by *scatter_from* and *scatter_to*

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`: A future referring to 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**(char **const** **basename*, *std*::vector<*T*> **const** &*result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1))

Scatter (send) a part of the value set at the given call site

This function transmits the value given by *result* to a central scatter site (where the corresponding *scatter_from* is executed)

Note Each scatter operation has to be accompanied with a unique usage of the *HPX_REGISTER_SCATTER* macro to define the necessary internal facilities used by *scatter_from* and *scatter_to*

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**(char **const** **basename*, *std*::vector<*T*> &&*result*, *std*::size_t *num_sites* = *std*::size_t(-1), *std*::size_t *generation* = *std*::size_t(-1), *std*::size_t *this_site* = *std*::size_t(-1))

Header `hpx/collectives/spmd_block.hpp`

namespace hpx

namespace lcos

Functions

template<typename **F**, typename ...**Args**>
hpx::future<void> **define_spmd_block** (*std*::string &&*name*, *std*::size_t *images_per_locality*, *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 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<*util*::all_of<**typename** *std*::is_integral<*I*>::type...>::value>::type **sync_images** (*I*... *i*)

hpx::future<void> **sync_images** (*hpx*::*launch*::async_policy **const** &*policy*, *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<*traits*::is_input_iterator<*Iterator*>::value, *hpx*::future<void>>::type **sync_images** (*hpx*::*launch*::async
const
&policy, *Iterator begin*, *Iterator end*) **const**

template<typename ...**I**>
std::enable_if<*util*::all_of<**typename** *std*::is_integral<*I*>::type...>::value, *hpx*::future<void>>::type **sync_images** (

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_**

Friends

friend hpx::lcos::hpx::serialization::access

Header `hpx/distributed/barrier.hpp`

Header `hpx/distributed/latch.hpp`

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.

Header `hpx/command_line_handling/command_line_handling.hpp`

namespace hpx

namespace util

Functions

std::size_t **get_num_high_priority_queues**(*util*::*command_line_handling* **const** &*cfg*, *std*::size_t *num_threads*)

std::string **get_affinity_domain**(*util*::*command_line_handling* **const** &*cfg*)

std::size_t **get_affinity_description**(*util*::*command_line_handling* **const** &*cfg*, *std*::string &*affinity_desc*)

std::size_t **get_pu_offset**(*util*::*command_line_handling* **const** &*cfg*)

std::size_t **get_pu_step**(*util*::*command_line_handling* **const** &*cfg*)

struct command_line_handling

#include <command_line_handling.hpp>

Public Functions

command_line_handling()

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_**

int **parse_result_**

Protected Functions

bool **handle_arguments** (*util*::*manage_config* &*cfgmap*, *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*)

Header hpx/command_line_handling/parse_command_line.hpp

namespace hpx

namespace util

Enums

enum commandline_error_mode

Values:

return_on_error

rethrow_on_error

allow_unregistered

report_missing_config_file = 0x80

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*, *std*::string **const** &*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)

std::string **reconstruct_command_line**(*hpx*::*program_options*::*variables_map* **const** &*vm*)

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.

Header `hpx/components_base/get_lva.hpp`

template<typename **Component**>
struct get_lva<*Component*, **typename** *std*::enable_if<!*traits*::*is_managed_component*<*Component*>::value>::type>
#include <get_lva.hpp>

Public Static Functions

static Component ***call**(*naming*::*address_type lva*)

template<typename **Component**>
struct get_lva<*Component*, **typename** *std*::enable_if<*traits*::*is_managed_component*<*Component*>::value && !*std*::is_const<C
#include <get_lva.hpp>

Public Static Functions

static Component ***call**(*naming*::*address_type lva*)

template<typename **Component**>
struct get_lva<*Component*, **typename** *std*::enable_if<*traits*::*is_managed_component*<*Component*>::value && *std*::is_const<C
#include <get_lva.hpp>

Public Static Functions

static Component ***call**(*naming*::*address_type lva*)

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.

template<typename **Component**>
struct get_lva<*Component*, **typename** *std*::enable_if<!*traits*::*is_managed_component*<*Component*>::value>::type>
#include <get_lva.hpp>

Public Static Functions

static Component ***call**(*naming*::*address_type lva*)

template<typename **Component**>
struct get_lva<*Component*, **typename** *std*::enable_if<*traits*::*is_managed_component*<*Component*>::value && !*std*::is_c
#include <get_lva.hpp>

Public Static Functions

static Component ***call**(*naming*::*address_type lva*)

template<typename **Component**>
struct get_lva<*Component*, **typename** *std*::enable_if<*traits*::*is_managed_component*<*Component*>::value && *std*::is_co
#include <get_lva.hpp>

Public Static Functions

static Component ***call**(*naming*::*address_type lva*)

Header hpx/components_base/pinned_ptr.hpp

template<typename **Component**>
struct create_helper<*Component*, **typename** *std*::enable_if<*traits*::component_decorates_action<*Component*>::value>::type

Public Static Functions

static *pinned_ptr* **call**(*naming*::*address_type lva*)

namespace hpx

namespace components

class pinned_ptr
#include <pinned_ptr.hpp>

Public Functions

pinned_ptr()

pinned_ptr(*pinned_ptr* **const** &*rhs*)

pinned_ptr(*pinned_ptr* &&*rhs*)

pinned_ptr &**operator=**(*pinned_ptr* **const** &*rhs*)

pinned_ptr &**operator=**(*pinned_ptr* &&*rhs*)

Public Static Functions

template<typename **Component**>
static *pinned_ptr* **create** (*naming*::*address_type lva*)

Private Functions

template<typename **Component**>
pinned_ptr (*naming*::*address_type lva*, id<*Component*>)

Private Members

std::unique_ptr<detail::pinned_ptr_base> **data_**

template<typename **Component**, typename **Enable** = void>
struct create_helper

Public Static Functions

static *pinned_ptr* **call** (*naming*::*address_type*)

template<typename **Component**>
struct create_helper<*Component*, **typename** *std*::enable_if<*traits*::component_decorates_action<*Componen*

Public Static Functions

static *pinned_ptr* **call** (*naming*::*address_type lva*)

Header `hpx/components_base/traits/component_pin_support.hpp`

namespace hpx

namespace traits

template<typename **Component**, typename **Enable** = void>
struct component_pin_support
#include <component_pin_support.hpp>

Public Static Functions

static constexpr void **pin** (Component **p*)

static constexpr bool **unpin** (Component **p*)

static constexpr *std*::uint32_t **pin_count** (Component **p*)

Header `hpx/components_base/traits/is_component.hpp`

namespace hpx

namespace traits

template<typename **Component**>
struct is_fixed_component : **public** *std*::integral_constant<bool, *std*::is_base_of<*traits*::detail::fixed_componen
#include <is_component.hpp> 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_co
#include <is_component.hpp> Subclassed by hpx::traits::is_managed_component< Component const >

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.

Header `hpx/compute/host.hpp`

Header `hpx/compute/host/block_allocator.hpp`

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*::*parallel*::*execution*::p
#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*::*parallel*::*execution*::parallel_policy_shim<*executor_type*, *executor_parameters_ty*

template<>
using base_type = detail::policy_allocator<T, *policy_type*>

template<>
using target_type = *std*::vector<*host*::*target*>

Public Functions

block_allocator()

block_allocator(target_type **const** &*targets*)

block_allocator(target_type &&*targets*)

target_type **const** &**target**() **const**

Header `hpx/compute/host/block_executor.hpp`

template<typename **Executor**>
struct executor_execution_category<*compute*::*host*::block_executor<*Executor*>>
#include <block_executor.hpp>

Public Types

typedef *parallel*::*execution*::*parallel_execution_tag* **type**

namespace hpx

namespace compute

namespace host

template<typename **Executor** = *hpx*::*parallel*::*execution*::*restricted_thread_pool_executor*>
struct block_executor
#include <block_executor.hpp> The block executor can be used to build NUMA aware programs. It will distribute work evenly across the passed targets

Template Parameters
- `Executor`: The underlying executor to use

Public Types

typedef *hpx::parallel::execution::static_chunk_size* **executor_parameters_type**

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>> **bu**

template<typename **F**, typename **Shape**, typename ...**Ts**>
parallel::*execution*::detail::bulk_execute_result<*F*, *Shape*, *Ts*...>::type **bulk_sync_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*>>
#include <block_executor.hpp>

Public Types

typedef *parallel*::*execution*::*parallel_execution_tag* **type**

Header hpx/compute/host/get_targets.hpp

namespace hpx

namespace compute

namespace host

Functions

std::vector<*target*> **get_local_targets** ()

hpx::future<*std*::vector<*target*>> **get_targets** (*hpx*::id_type **const** &*locality*)

Header hpx/compute/host/numa_allocator.hpp

namespace hpx

namespace parallel

namespace util

template<typename **T**, typename **Executors**>
class numa_allocator
#include <numa_allocator.hpp>

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*, **typename** *std*::allocator<void>::const_pointer = 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** &*l*, numa_allocator **const** &*r*)

template<typename **U**>
struct rebind
#include <numa_allocator.hpp>

Public Types

template<>
typedef numa_allocator<U, Executors> **other**

Header `hpx/compute/host/numa_binding_allocator.hpp`

Defines

NUMA_BINDING_ALLOCATOR_DEBUG

namespace hpx

Functions

static hpx::debug::enable_print<NUMA_BINDING_ALLOCATOR_DEBUG> hpx::nba_deb("NUM_B_A")

namespace compute

namespace host

Typedefs

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 const T ***const_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

std::shared_ptr<numa_binding_helper<T>> **binding_helper_**

threads::*hpx_hwloc_membind_policy* **policy_**

unsigned int **flags_**

Protected Functions

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

std::mutex **init_mutex**

template<typename **U**>
struct rebind
#include <numa_binding_allocator.hpp>

Public Types

template<>
typedef numa_binding_allocator<U> **other**

template<typename **T**>
struct numa_binding_helper
#include <numa_binding_allocator.hpp>

Public Functions

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 *axis*) **const**

virtual *std*::size_t **memory_step** (*std*::size_t *axis*) **const**

virtual *std*::size_t **display_step** (*std*::size_t *axis*) **const**

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>
#include <numa_binding_allocator.hpp>

Public Functions

int **operator()** (int **const** &*domain*) **const**

Header `hpx/compute/host/numa_domains.hpp`

namespace hpx

namespace compute

namespace host

Functions

std::vector<*target*> **numa_domains**()

Header `hpx/compute/host/target.hpp`

namespace hpx

namespace compute

namespace host

struct target
#include <target.hpp>

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**

hpx::future<void> **get_future**() **const**

Public Static Functions

static *std*::vector<*target*> **get_local_targets**()

static *hpx*::future<*std*::vector<*target*>> **get_targets**(*hpx*::id_type **const** &*locality*)

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
#include <target.hpp>

Public Functions

native_handle_type()

native_handle_type(*hpx*::*threads*::mask_type *mask*)

hpx::*threads*::mask_type &**get_device**()

hpx::*threads*::mask_type **const** &**get_device**() **const**

Private Members

hpx::*threads*::mask_type **mask_**

Friends

friend hpx::compute::host::target

Header `hpx/compute/host/target_distribution_policy.hpp`

Header `hpx/compute/host/traits/access_target.hpp`

template<>
struct access_target<*host*::*target*>
#include <access_target.hpp>

Public Types

typedef *host::target* **target_type**

Public Static Functions

template<typename **T**>
static *T* **const** &**read**(*target_type* **const** &*tgt*, *T* **const** **t*)

template<typename **T**>
static void **write**(*target_type* **const** &*tgt*, *T* **dst*, *T* **const** **src*)

template<>
struct access_target<*std*::vector<*host*::*target*>>
#include <access_target.hpp>

Public Types

typedef *std*::vector<*host*::*target*> **target_type**

Public Static Functions

template<typename **T**>
static *T* **const** &**read**(*target_type* **const** &*tgt*, *T* **const** **t*)

template<typename **T**>
static void **write**(*target_type* **const** &*tgt*, *T* **dst*, *T* **const** **src*)

namespace hpx

namespace compute

namespace traits

template<>
struct access_target<*host*::*target*>
#include <access_target.hpp>

Public Types

typedef *host::target* **target_type**

Public Static Functions

template<typename **T**>
static *T* **const** &**read**(*target_type* **const** &*tgt*, *T* **const** **t*)

template<typename **T**>
static void **write**(*target_type* **const** &*tgt*, *T* **dst*, *T* **const** **src*)

template<>
struct access_target<*std*::vector<*host*::*target*>>
#include <access_target.hpp>

Public Types

typedef *std*::vector<*host*::*target*> **target_type**

Public Static Functions

template<typename **T**>
static *T* **const** &**read**(*target_type* **const** &*tgt*, *T* **const** **t*)

template<typename **T**>
static void **write**(*target_type* **const** &*tgt*, *T* **dst*, *T* **const** **src*)

Header `hpx/compute/serialization/vector.hpp`

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)

Header `hpx/compute/traits.hpp`

Header `hpx/compute/traits/access_target.hpp`

Header `hpx/compute/traits/allocator_traits.hpp`

namespace hpx

namespace compute

namespace traits

template<typename **Allocator**>
struct allocator_traits : **public** *std*::allocator_traits<*Allocator*>
#include <allocator_traits.hpp>

Public Types

typedef detail::get_reference_type<Allocator>::type **reference**

typedef detail::get_const_reference_type<Allocator>::type **const_reference**

typedef detail::get_target_traits<Allocator>::type **access_target**

typedef *access_target*::target_type **target_type**

Public Static Functions

static auto **target** (Allocator **const** &*alloc*)

template<typename ...**Ts**>
static void **bulk_construct** (Allocator &*alloc*, pointer *p*, size_type *count*, *Ts*&&... *vs*)

static void **bulk_destroy** (Allocator &*alloc*, pointer *p*, size_type *count*)

Private Types

typedef *std*::allocator_traits<Allocator> **base_type**

Header `hpx/compute/vector.hpp`

namespace hpx

namespace compute

Functions

template<typename **T**, typename **Allocator**>
void **swap** (vector<*T*, *Allocator*> &*x*, vector<*T*, *Allocator*> &*y*)
Effects: x.swap(y);.

template<typename **T**, typename **Allocator** = *std*::allocator<*T*>>
class vector
#include <vector.hpp>

Public Types

typedef T **value_type**
Member types (FIXME: add reference to std.

typedef Allocator **allocator_type**

typedef *alloc_traits*::access_target **access_target**

typedef *std*::size_t **size_type**

typedef *std*::ptrdiff_t **difference_type**

typedef *alloc_traits*::reference **reference**

typedef *alloc_traits*::const_reference **const_reference**

typedef *alloc_traits*::pointer **pointer**

typedef *alloc_traits*::const_pointer **const_pointer**

typedef detail::iterator<T, Allocator> **iterator**

typedef detail::iterator<T **const**, Allocator> **const_iterator**

typedef detail::reverse_iterator<T, Allocator> **reverse_iterator**

typedef detail::const_reverse_iterator<T, Allocator> **const_reverse_iterator**

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*>::v
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**
Returns the allocator associated with the container.

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 size*)
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 size*, T **const** &*val*)
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.

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_**

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.

Header `hpx/compute/cuda.hpp`

Header `hpx/compute/cuda/allocator.hpp`

Header `hpx/compute/cuda/concurrent_executor.hpp`

Header `hpx/compute/cuda/concurrent_executor_parameters.hpp`

Header `hpx/compute/cuda/default_executor.hpp`

Header `hpx/compute/cuda/default_executor_parameters.hpp`

Header `hpx/compute/cuda/serialization/value_proxy.hpp`

Header `hpx/compute/cuda/target_distribution_policy.hpp`

Header `hpx/compute/cuda/target_ptr.hpp`

Header `hpx/compute/cuda/traits/access_target.hpp`

Header `hpx/compute/cuda/transfer.hpp`

Header `hpx/compute/cuda/value_proxy.hpp`

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.

Header `hpx/concepts/concepts.hpp`

Defines

HPX_CONCEPT_REQUIRES_(...)

HPX_CONCEPT_REQUIRES(...)

HPX_CONCEPT_ASSERT(...)

Header `hpx/concepts/has_member_xxx.hpp`

Defines

HPX_HAS_MEMBER_XXX_TRAIT_DEF(*MEMBER*)

This macro creates a boolean unary metafunction 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.

Header `hpx/concepts/has_xxx.hpp`

Defines

HPX_HAS_XXX_TRAIT_DEF(*Name*)

This macro creates a boolean unary metafunction 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.

Header `hpx/concurrency/barrier.hpp`

namespace hpx

namespace util

class barrier

#include <barrier.hpp>

Public Functions

barrier (*std*::size_t *number_of_threads*)

~barrier ()

void **wait** ()

Private Types

typedef *std*::mutex **mutex_type**

Private Members

std::size_t **const number_of_threads_**

std::size_t **total_**

mutex_type **mtx_**

std::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)

Header `hpx/concurrency/cache_line_data.hpp`

template<typename **Data**>
struct cache_aligned_data<*Data*, *std*::false_type>

#include <cache_line_data.hpp>

Public Functions

cache_aligned_data()

cache_aligned_data(Data &&*data*)

cache_aligned_data(Data **const** &*data*)

Public Members

Data **data_**

template<typename **Data**>
struct cache_aligned_data_derived<*Data*, *std*::false_type> : **public** *Data*
#include <cache_line_data.hpp>

Public Functions

cache_aligned_data_derived()

cache_aligned_data_derived(Data &&*data*)

cache_aligned_data_derived(Data **const** &*data*)

namespace hpx

namespace threads

Functions

constexpr *std*::size_t **get_cache_line_size**()

namespace util

Typedefs

using cache_line_data = cache_aligned_data<Data>

template<typename **Data**, typename **NeedsPadding** = **typename** detail::needs_padding<*Data*>::type>
struct cache_aligned_data
#include <cache_line_data.hpp>

Public Functions

cache_aligned_data()

cache_aligned_data(Data &&*data*)

cache_aligned_data(Data **const** &*data*)

Public Members

Data **data_**

template<>
char **cacheline_pad**[**get_cache_line_padding_size**(sizeof(Data))]

template<typename **Data**>
struct cache_aligned_data<*Data*, *std*::false_type>
#include <cache_line_data.hpp>

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*
#include <cache_line_data.hpp>

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*
#include <cache_line_data.hpp>

Public Functions

cache_aligned_data_derived()

cache_aligned_data_derived(Data &&*data*)

cache_aligned_data_derived(Data **const** &*data*)

Header `hpx/concurrency/concurrentqueue.hpp`

Defines

MOODYCAMEL_THREADLOCAL

MOODYCAMEL_EXCEPTIONS_ENABLED

MOODYCAMEL_TRY

MOODYCAMEL_CATCH(...)

MOODYCAMEL_RETHROW

MOODYCAMEL_THROW(*expr*)

MOODYCAMEL_NOEXCEPT

MOODYCAMEL_NOEXCEPT_CTOR(*type*, *valueType*, *expr*)

MOODYCAMEL_NOEXCEPT_ASSIGN(*type*, *valueType*, *expr*)

MOODYCAMEL_DELETE_FUNCTION

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
#include <concurrentqueue.hpp>

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*)

template<typename **U**>
bool **try_dequeue_non_interleaved** (*U &item*)

template<typename **U**>
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_BLOC

const *size_t* **EXPLICIT_INITIAL_INDEX_SIZE** = static_cast<*size_t*>(Traits::EXPLICIT_INITIAL_INDEX_SIZ

const *size_t* **IMPLICIT_INITIAL_INDEX_SIZE** = static_cast<*size_t*>(Traits::IMPLICIT_INITIAL_INDEX_SIZ

const *size_t* **INITIAL_IMPLICIT_PRODUCER_HASH_SIZE** = static_cast<*size_t*>(Traits::INITIAL_IMPLICIT_

const *std*::uint32_t **EXPLICIT_CONSUMER_CONSUMPTION_QUOTA_BEFORE_ROTATE** = static_cast<*std*::uint3

const size_t hpx::concurrency::ConcurrentQueue::MAX_SUBQUEUE_SIZE = (details::co

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_implicit_producer_hashes**(ConcurrentQueue &*other*)

ImplicitProducer ***get_or_add_implicit_producer**()

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*> **initialImplicitProducer**

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*, *XTraits*>::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::empt

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

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

template<>
void **add_knowing_refcount_is_zero** (N **node*)

Private Members

template<>
std::atomic<N*> **freeListHead**

Private Static Attributes

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
#include <concurrentqueue.hpp>

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

#include <concurrentqueue.hpp>

Public Functions

template<typename **T**, typename **Traits**>
ConsumerToken(ConcurrentQueue<*T*, *Traits*> &*q*)

template<typename **T**, typename **Traits**>
ConsumerToken(BlockingConcurrentQueue<*T*, *Traits*> &*q*)

ConsumerToken(*ConsumerToken* &&*other*)

ConsumerToken &**operator=**(*ConsumerToken* &&*other*)

void **swap**(*ConsumerToken* &*other*)

ConsumerToken(*ConsumerToken* **const**&)

ConsumerToken &**operator=**(*ConsumerToken* **const**&)

Private Members

std::uint32_t **initialOffset**

std::uint32_t **lastKnownGlobalOffset**

std::uint32_t **itemsConsumedFromCurrent**

details::*ConcurrentQueueProducerTypelessBase* ***currentProducer**

details::*ConcurrentQueueProducerTypelessBase* ***desiredProducer**

Friends

friend hpx::concurrency::ConcurrentQueue

friend hpx::concurrency::ConcurrentQueueTests

struct ProducerToken

#include <concurrentqueue.hpp>

Public Functions

template<typename **T**, typename **Traits**>
ProducerToken (ConcurrentQueue<*T*, *Traits*> &*queue*)

template<typename **T**, typename **Traits**>
ProducerToken (BlockingConcurrentQueue<*T*, *Traits*> &*queue*)

ProducerToken (*ProducerToken* &&*other*)

ProducerToken &**operator=** (*ProducerToken* &&*other*)

void **swap** (*ProducerToken* &*other*)

bool **valid** () **const**

~ProducerToken ()

ProducerToken (*ProducerToken* **const**&)

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

const *thread_id_t* **invalid_thread_id** = 0

const *thread_id_t* **invalid_thread_id2** = 1

template<bool **use32**>
struct _hash_32_or_64

#include <concurrentqueue.hpp>

Public Static Functions

static *std*::uint32_t **hash**(*std*::uint32_t *h*)

template<>
struct _hash_32_or_64<1>
#include <concurrentqueue.hpp>

Public Static Functions

static *std*::uint64_t **hash**(*std*::uint64_t *h*)

struct ConcurrentQueueProducerTypelessBase
#include <concurrentqueue.hpp>

Public Functions

ConcurrentQueueProducerTypelessBase()

Public Members

ConcurrentQueueProducerTypelessBase ***next**

std::atomic<bool> **inactive**

ProducerToken ***token**

template<typename **T**>
struct const_numeric_max
#include <concurrentqueue.hpp>

Public Static Attributes

const T hpx::concurrency::details::const_numeric_max::value= std::numeric_lim

union max_align_t
#include <concurrentqueue.hpp>

Public Members

std_max_align_t **x**

long long **y**

void ***z**

template<bool **Enable**>
struct nomove_if
#include <concurrentqueue.hpp>

Public Static Functions

template<typename **T**>
static *T* **const** &**eval** (*T* **const** &*x*)

template<>
struct nomove_if<false>
#include <concurrentqueue.hpp>

Public Static Functions

template<typename **U**>
static auto **eval** (*U* &&*x*)

template<>
struct static_is_lock_free<bool>
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = ATOMIC_BOOL_LOCK_FREE

template<typename **U**>
struct static_is_lock_free<*U**>
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = ATOMIC_POINTER_LOCK_FREE

template<typename **T**>
struct static_is_lock_free_num
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = 0

template<>
struct static_is_lock_free_num<int>
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = ATOMIC_INT_LOCK_FREE

template<>
struct static_is_lock_free_num<long>
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = ATOMIC_LONG_LOCK_FREE

template<>
struct static_is_lock_free_num<long long>
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = ATOMIC_LLONG_LOCK_FREE

template<>
struct static_is_lock_free_num<short>
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = ATOMIC_SHORT_LOCK_FREE

template<>
struct static_is_lock_free_num<signed char>
#include <concurrentqueue.hpp>

Public Types

enum [anonymous]
Values:

value = ATOMIC_CHAR_LOCK_FREE

template<typename **thread_id_t**>
struct thread_id_converter
#include <concurrentqueue.hpp>

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*)

Header `hpx/concurrency/deque.hpp`

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

#include <deque.hpp>

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<*nod*

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 **const** &*data*)

bool **push_right** (T **const** &*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)

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

constexpr *std*::size_t **padding_size** = BOOST_LOCKFREE_CACHELINE_BYTES - sizeof(anchor)

template<typename **T**>
struct deque_anchor

#include <deque.hpp>

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**() **volatile const**

node ***left**() **volatile const**

node ***right**() **volatile const**

tag_t **status**() **volatile const**

tag_t **tag**() **volatile const**

bool **cas**(deque_anchor &*expected*, deque_anchor **const** &*desired*) **volatile**

bool **cas**(pair &*expected*, deque_anchor **const** &*desired*) **volatile**

bool **cas**(deque_anchor &*expected*, pair **const** &*desired*) **volatile**

bool **cas**(pair &*expected*, pair **const** &*desired*) **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

#include <deque.hpp>

Public Types

typedef detail::tagged_ptr<deque_node> **pointer**

typedef *std*::atomic<*pointer*> **atomic_pointer**

typedef *pointer*::tag_t **tag_t**

Public Functions

deque_node()

deque_node(*deque_node* **const** &*p*)

deque_node(*deque_node* **lptr*, *deque_node* **rptr*, T **const** &*v*, *tag_t ltag* = 0, *tag_t rtag* = 0)

Public Members

atomic_pointer **left**

atomic_pointer **right**

T **data**

Header hpx/concurrency/spinlock.hpp

namespace hpx

namespace util

struct spinlock

#include <spinlock.hpp> boost::mutex-compatible spinlock class

Public Types

typedef *boost*::detail::spinlock ***native_handle_type**

Public Functions

HPX_NON_COPYABLE(*spinlock*)

spinlock(char **const*** = nullptr)

~spinlock()

void **lock**()

bool **try_lock**()

void **unlock**()

native_handle_type **native_handle**()

Private Members

boost::detail::spinlock **m** = BOOST_DETAIL_SPINLOCK_INIT

Header `hpx/concurrency/spinlock_pool.hpp`

namespace hpx

namespace util

template<typename **Tag**, *std*::size_t **N** = HPX_HAVE_SPINLOCK_POOL_NUM>
class spinlock_pool
#include <spinlock_pool.hpp>

Public Static Functions

static *boost*::detail::spinlock &**spinlock_for**(void **const** **pv*)

Private Static Attributes

cache_aligned_data<*boost*::detail::spinlock> **pool_**

class scoped_lock
#include <spinlock_pool.hpp>

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<>
boost::detail::spinlock &**sp_**

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.

Header `hpx/config.hpp`

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 = ...

(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 changes 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 5 levels of stack back trace on suspension

HPX_MAX_NETWORK_RETRIES

HPX_NETWORK_RETRIES_SLEEP

HPX_INI_PATH_DELIMITER

HPX_PATH_DELIMITERS

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_APPLICATION_STRING

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_IDLE_BACKOFF_TIME_MAX

HPX_WRAPPER_HEAP_STEP

HPX_INITIAL_GID_RANGE

HPX_CONTINUATION_MAX_RECURSION_DEPTH

HPX_AGAS_BOOTSTRAP_PREFIX

HPX_AGAS_NS_MSB

HPX_AGAS_PRIMARY_NS_MSB

HPX_AGAS_PRIMARY_NS_LSB

HPX_AGAS_COMPONENT_NS_MSB

HPX_AGAS_COMPONENT_NS_LSB

HPX_AGAS_SYMBOL_NS_MSB

HPX_AGAS_SYMBOL_NS_LSB

HPX_AGAS_LOCALITY_NS_MSB

HPX_AGAS_LOCALITY_NS_LSB

Header `hpx/config/asio.hpp`

Header `hpx/config/attributes.hpp`

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 __.

Header hpx/config/autolink.hpp

Header hpx/config/branch_hints.hpp

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.

Header hpx/config/compiler_fence.hpp

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.

Header hpx/config/compiler_native_tls.hpp

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.

Header hpx/config/compiler_specific.hpp

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.

Header `hpx/config/constexpr.hpp`

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` if the compiler supports it, `constexpr` otherwise.

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.

Header `hpx/config/debug.hpp`

Defines

HPX_DEBUG
Defined if HPX is compiled in debug mode.

HPX_BUILD_TYPE
Evaluates to `debug` if compiled in debug mode, `release` otherwise.

Header `hpx/config/deprecation.hpp`

Defines

HPX_HAVE_DEPRECATION_WARNINGS_V1_4

HPX_DEPRECATED_V1_4 (*x*)

HPX_HAVE_DEPRECATION_WARNINGS_V1_5

HPX_DEPRECATED_V1_5 (*x*)

HPX_HAVE_DEPRECATION_WARNINGS_V1_6

HPX_DEPRECATED_V1_6 (*x*)

HPX_DEPRECATED_V (*major*, *minor*, *x*)

Header hpx/config/emulate_deleted.hpp

Defines

HPX_NON_COPYABLE (*cls*)
Marks a class as non-copyable and non-movable.

Header hpx/config/export_definitions.hpp

Defines

HPX_EXPORT
Marks a class or function to be exported from HPX or imported if it is consumed.

Header hpx/config/forceinline.hpp

Defines

HPX_FORCEINLINE
Marks a function to be forced inline.

Header hpx/config/lambda_capture.hpp

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.

Header hpx/config/manual_profiling.hpp

Defines

HPX_SUPER_PURE

HPX_PURE

HPX_HOT

HPX_COLD

Header hpx/config/threads_stack.hpp

Defines

HPX_THREADS_STACK_OVERHEAD

HPX_SMALL_STACK_SIZE

HPX_MEDIUM_STACK_SIZE

HPX_LARGE_STACK_SIZE

HPX_HUGE_STACK_SIZE

Header hpx/config/warnings_prefix.hpp

Header hpx/config/warnings_suffix.hpp

Header hpx/config/weak_symbol.hpp

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.

Header hpx/modules/config_registry.hpp

namespace hpx

namespace config_registry

Functions

std::vector<*module_config*> **const** &**get_module_configs**()

void **add_module_config**(*module_config* **const** &*config*)

struct add_module_config_helper
#include <config_registry.hpp>

Public Functions

add_module_config_helper (*module_config* **const** &*config*)

struct module_config
#include <config_registry.hpp>

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.

Header `hpx/coroutines/coroutine.hpp`

namespace hpx

namespace threads

namespace coroutines

class coroutine
#include <coroutine.hpp>

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 **rebind**(*functor_type* &&*f*, *thread_id_type* *id*)

result_type **operator()**(*arg_type* *arg* = *arg_type*())

bool **is_ready**() **const**

std::ptrdiff_t **get_available_stack_space**()

impl_type ***impl**()

Private Members

impl_type **impl_**

Header `hpx/coroutines/coroutine_fwd.hpp`

Header `hpx/coroutines/stackless_coroutine.hpp`

namespace hpx

namespace threads

namespace coroutines

class stackless_coroutine

#include <stackless_coroutine.hpp>

Public Types

using thread_id_type = *hpx*::*threads*::*thread_id*

using result_type = *std*::pair<*thread_state_enum*, *thread_id_type*>

using arg_type = *thread_state_ex_enum*

using functor_type = *util*::*unique_function_nonser*<*result_type* (*arg_type*) >

Public Functions

stackless_coroutine (*functor_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 **rebind** (*functor_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_**

Header hpx/coroutines/thread_enums.hpp

namespace hpx

namespace threads

Enums

enum thread_state_enum

The *thread_state_enum* 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

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

enum thread_priority

This enumeration lists all possible thread-priorities for HPX threads.

Values:

thread_priority_unknown = -1

thread_priority_default = 0
Will assign the priority of the task to the default (normal) priority.

thread_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.

thread_priority_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.

thread_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.

thread_priority_boost = 4
Same as *thread_priority_high* except that the thread will fall back to *thread_priority_normal* if resumed after being suspended.

thread_priority_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).

thread_priority_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.

enum thread_state_ex_enum
The *thread_state_ex_enum* enumerator encodes the reason why a thread is being restarted

Values:

wait_unknown = 0

wait_signaled = 1
The thread has been signaled.

wait_timeout = 2
The thread has been reactivated after a timeout

wait_terminate = 3
The thread needs to be terminated.

wait_abort = 4
The thread needs to be aborted.

enum thread_stacksize
A *thread_stacksize* references any of the possible stack-sizes for HPX threads.

Values:

thread_stacksize_unknown = -1

thread_stacksize_small = 1
use small stack size

thread_stacksize_medium = 2
use medium sized stack size

thread_stacksize_large = 3
use large stack size

thread_stacksize_huge = 4
use very large stack size

thread_stacksize_nostack = 5
this thread does not suspend (does not need a stack)

thread_stacksize_current = 6
use size of current thread's stack

thread_stacksize_default = *thread_stacksize_small*
use default stack size

thread_stacksize_minimal = *thread_stacksize_small*
use minimally stack size

thread_stacksize_maximal = *thread_stacksize_huge*
use maximally stack size

enum thread_schedule_hint_mode
The type of hint given when creating new tasks.

Values:

thread_schedule_hint_mode_none = 0

thread_schedule_hint_mode_thread = 1

thread_schedule_hint_mode_numa = 2

Functions

char **const** ***get_thread_state_name**(*thread_state_enum 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.

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.

char **const** ***get_thread_state_ex_name**(*thread_state_ex_enum state*)
Get the readable string representing the name of the given thread_state_ex_enum constant.

char **const** ***get_thread_state_name**(thread_state *state*)
Get the readable string representing the name of the given thread_state constant.

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

struct thread_schedule_hint
#include <thread_enums.hpp>

Public Functions

constexpr thread_schedule_hint()

constexpr thread_schedule_hint(*std*::int16_t *thread_hint*)

constexpr thread_schedule_hint(*thread_schedule_hint_mode mode*, *std*::int16_t *hint*)

bool **operator==**(*thread_schedule_hint* **const** &*rhs*) **const**

bool **operator!=**(*thread_schedule_hint* **const** &*rhs*) **const**

Public Members

thread_schedule_hint_mode **mode**

std::int16_t **hint**

Header hpx/coroutines/thread_id_type.hpp

namespace hpx

namespace threads

Variables

constexpr *thread_id* **invalid_thread_id**

struct thread_id
#include <thread_id_type.hpp>

Public Functions

constexpr thread_id()

constexpr thread_id(*thread_id_repr thrd*)

thread_id(*thread_id* **const**&)

thread_id &**operator=**(*thread_id* **const**&)

thread_id(*thread_id* &&*rhs*)

thread_id &**operator=**(*thread_id* &&*rhs*)

constexpr operator bool() **const**

constexpr *thread_id_repr* **get**() **const**

constexpr void **reset**()

Private Types

using thread_id_repr = void*

Private Members

thread_id_repr **thrd_**

Friends

constexpr bool **operator==**(*std*::nullptr_t, thread_id **const** &*rhs*)

constexpr bool **operator!=**(*std*::nullptr_t, thread_id **const** &*rhs*)

constexpr bool **operator==**(thread_id **const** &*lhs*, *std*::nullptr_t)

constexpr bool **operator!=**(thread_id **const** &*lhs*, *std*::nullptr_t)

constexpr bool **operator==**(thread_id **const** &*lhs*, thread_id **const** &*rhs*)

constexpr bool **operator!=**(thread_id **const** &*lhs*, thread_id **const** &*rhs*)

constexpr bool **operator<**(thread_id **const** &*lhs*, thread_id **const** &*rhs*)

constexpr bool **operator>**(thread_id **const** &*lhs*, thread_id **const** &*rhs*)

constexpr bool **operator<=**(thread_id **const** &*lhs*, thread_id **const** &*rhs*)

constexpr bool **operator>=**(thread_id **const** &*lhs*, thread_id **const** &*rhs*)

template<typename **Char**, typename **Traits**>
std::basic_ostream<*Char*, *Traits*> &**operator<<**(*std*::basic_ostream<*Char*, *Traits*> &*os*, thread_id **const** &*id*)

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.

Header `hpx/datastructures/any.hpp`

template<>
class basic_any<void, void, void, *std*::true_type>
#include <any.hpp>

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*>::ty
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_copy_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

~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*>::ty
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<void, void, *Char*, *std*::true_type>
#include <any.hpp>

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*>::ty
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_copy_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

~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*>::ty
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>
#include <any.hpp>

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*>::ty
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_move_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

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*>::ty
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>
#include <any.hpp>

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*>::ty
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_move_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

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*>::ty
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

namespace util

Typedefs

using any_nonser = *basic_any*<void, void, void, *std*::true_type>

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 unique_any_nonser = *basic_any*<void, void, void, *std*::false_type>

using streamable_unique_any_nonser = basic_any<void, void, char, *std*::false_type>

using streamable_unique_wany_nonser = basic_any<void, void, wchar_t, *std*::false_type>

Functions

template<typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**, typename **Enable** = **typename**
std::basic_istream<*Char*> &**operator>>**(*std*::basic_istream<*Char*> &*i*, basic_any<*IArch*, *OArch*, *Char*, *Copyable*> &*obj*)

template<typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**, typename **Enable** = **typename**
std::basic_ostream<*Char*> &**operator<<**(*std*::basic_ostream<*Char*> &*o*, basic_any<*IArch*, *OArch*, *Char*, *Copyable*> **const** &*obj*)

template<typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>

void **swap**(basic_any<*IArch*, *OArch*, *Char*, *Copyable*> &*lhs*, basic_any<*IArch*, *OArch*, *Char*, *Copyable*> &*rhs*)

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T ***any_cast**(basic_any<*IArch*, *OArch*, *Char*, *Copyable*> **operand*)

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T **const** ***any_cast**(basic_any<*IArch*, *OArch*, *Char*, *Copyable*> **const** **operand*)

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T **any_cast**(basic_any<*IArch*, *OArch*, *Char*, *Copyable*> &*operand*)

template<typename **T**, typename **IArch**, typename **OArch**, typename **Char**, typename **Copyable**>
T **const** &**any_cast**(basic_any<*IArch*, *OArch*, *Char*, *Copyable*> **const** &*operand*)

template<typename **T**>
basic_any<void, void, void, *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**>
basic_any<void, void, void, *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*)

struct bad_any_cast : **public** bad_cast
#include <any.hpp>

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**

template<typename **Char**>
class basic_any<void, void, *Char*, *std*::false_type>
#include <any.hpp>

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*::dec
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_move_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

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*::dec
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>
#include <any.hpp>

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*::dec
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_copy_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

~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*::dec
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>
#include <any.hpp>

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*::dec
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_move_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

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*::dec
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>
#include <any.hpp>

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*::dec
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_copy_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

~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*::dec
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*)

Header `hpx/datastructures/member_pack.hpp`

template<*std*::size_t... **Is**, typename ...**Ts**>
struct member_pack<*util*::*index_pack*<*Is*...>, *Ts*...> : **public** *hpx*::*util*::detail::member_leaf<*Is*, *Ts*>
#include <member_pack.hpp>

Public Functions

member_pack()

template<typename ...**Us**>
constexpr member_pack(*std*::piecewise_construct_t, *Us*&&... *us*)

template<*std*::size_t **I**>
decltype(auto) **constexpr get**() &

template<*std*::size_t **I**>
decltype(auto) **constexpr get**() **const** &

template<*std*::size_t **I**>
decltype(auto) **constexpr get**() &&

template<*std*::size_t **I**>
decltype(auto) **constexpr 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

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*>
#include <member_pack.hpp>

Public Functions

member_pack()

template<typename ...**Us**>
constexpr member_pack(*std*::piecewise_construct_t, *Us*&&... *us*)

template<*std*::size_t **I**>
decltype(auto) **constexpr get**() &

template<*std*::size_t **I**>
decltype(auto) **constexpr get**() **const** &

template<*std*::size_t **I**>
decltype(auto) **constexpr get**() &&

template<*std*::size_t **I**>
decltype(auto) **constexpr get**() **const** &&

Header hpx/datastructures/optional.hpp

template<typename **T**>
struct hash<*hpx*::*util*::optional<*T*>>
#include <optional.hpp>

Public Functions

constexpr *std*::size_t **operator()**(::*hpx*::*util*::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** &*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** &*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** &*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**>
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

constexpr *nullopt_t* **nullopt** = {*nullopt_t*::init()}

class bad_optional_access : **public** logic_error
#include <optional.hpp>

Public Functions

bad_optional_access (*std*::string **const** &*what_arg*)

bad_optional_access (char **const** **what_arg*)

struct nullopt_t
#include <optional.hpp>

Public Functions

constexpr nullopt_t (*nullopt_t*::init)

template<typename **T**>
class optional
#include <optional.hpp>

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*)

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**()

namespace std

template<typename **T**>
struct hash<*hpx*::*util*::optional<*T*>>
#include <optional.hpp>

Public Functions

constexpr *std*::size_t **operator()**(::*hpx*::*util*::optional<T> **const** &*arg*) **const**

Header `hpx/datastructures/tagged.hpp`

Defines

HPX_DEFINE_TAG_SPECIFIER(*NAME*)

namespace hpx

namespace util

template<typename **Base**, typename ...**Tags**>
struct tagged : **public** *Base*, **public** detail::getters::collect<*tagged*<*Base*, *Tags*...>, *Tags*...>
#include <tagged.hpp>

Public Functions

template<typename ...**Ts**>
tagged(*Ts&&... ts*)

template<typename **Other**>
tagged(*tagged<Other*, Tags...> *&&rhs*)

template<typename **Other**>
tagged(*tagged<Other*, Tags...> **const** *&rhs*)

template<typename **Other**>
tagged &**operator=**(tagged<*Other*, Tags...> *&&rhs*)

template<typename **Other**>
tagged &**operator=**(tagged<*Other*, Tags...> **const** *&rhs*)

template<typename **U**>
tagged &**operator=**(*U &&u*)

void **swap**(tagged *&other*)

Friends

void **swap**(tagged *&x*, tagged *&y*)

Header hpx/datastructures/tagged_pair.hpp

namespace hpx

namespace util

Functions

template<typename **Tag1**, typename **Tag2**, typename **T1**, typename **T2**>
constexpr tagged_pair<*Tag1* (**typename** decay<*T1*>::type), *Tag2*
typename decay<*T2*>::type> **make_tagged_pair***std*::pair<*T1*, *T2*> *&&p*

template<typename **Tag1**, typename **Tag2**, typename **T1**, typename **T2**>
constexpr tagged_pair<*Tag1* (**typename** decay<*T1*>::type), *Tag2*
typename decay<*T2*>::type> **make_tagged_pair***std*::pair<*T1*, *T2*> **const** *&p*

template<typename **Tag1**, typename **Tag2**, typename ...**Ts**>
constexpr tagged_pair<*Tag1* (**typename** tuple_element<0, tuple<*Ts*...>>::type), *Tag2*
typename tuple_element<1, tuple<*Ts*...>>::type> **make_tagged_pair**tuple<*Ts*...> *&&p*

template<typename **Tag1**, typename **Tag2**, typename ...**Ts**>
constexpr tagged_pair<*Tag1* (**typename** tuple_element<0, tuple<*Ts*...>>::type), *Tag2*
typename tuple_element<1, tuple<*Ts*...>>::type> **make_tagged_pair**tuple<*Ts*...> **const** *&p*

template<typename **Tag1**, typename **Tag2**, typename **T1**, typename **T2**>
constexpr tagged_pair<*Tag1* (**typename** decay<*T1*>::type), *Tag2*
typename decay<*T2*>::type> **make_tagged_pair***T1 &&t1*, *T2 &&t2*

template<typename **F**, typename **S**>
struct tagged_pair : **public** *hpx*::*util*::tagged<*std*::pair<detail::tag_elem<*F*>::type, detail::tag_elem<*S*>::type>, d
#include <tagged_pair.hpp>

Public Types

typedef tagged<*std*::pair<**typename** detail::tag_elem<F>::type, **typename** detail::tag_elem<S>::type>, **typena**

Public Functions

template<typename ...**Ts**>
tagged_pair(*Ts*&&... *ts*)

Header `hpx/datastructures/tagged_tuple.hpp`

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**(tuple<*Ts*...> &&*t*)

template<typename ...**Ts**>
struct tagged_tuple : **public** *hpx*::*util*::tagged<tuple<detail::tag_elem<*Ts*>::type...>, detail::tag_spec<*Ts*>::type..
#include <tagged_tuple.hpp>

Public Types

typedef tagged<tuple<**typename** detail::tag_elem<Ts>::type...>, **typename** detail::tag_spec<Ts>::type...> **base**

Public Functions

template<typename ...**Ts_**>
tagged_tuple(*Ts_*&&... *ts*)

Header hpx/datastructures/traits/is_tuple_like.hpp

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.

Header hpx/datastructures/traits/supports_streaming_with_any.hpp

Header hpx/datastructures/tuple.hpp

template<typename **T0**, typename **T1**>
struct tuple_element<0, *std*::pair<*T0*, *T1*>>
#include <tuple.hpp>

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*>>
#include <tuple.hpp>

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*>>
#include <tuple.hpp>

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

namespace util

Functions

template<typename ...**Ts**>
constexpr tuple<**typename** 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*)

template<typename ...**Ts**>
void **swap** (tuple<*Ts*...> &*x*, tuple<*Ts*...> &*y*)

Variables

detail::ignore_type **const ignore** = {}

template<typename ...**Ts**>
class tuple
#include <tuple.hpp>

Public Functions

template<typename **Dependent** = void, typename **Enable** = **typename** *std*::enable_if<*util*::all_of<*std*::is_construc
constexpr tuple ()

constexpr tuple (Ts **const**&... *vs*)

template<typename **U**, typename ...**Us**, typename **Enable** = **typename** *std*::enable_if<!*std*::is_same<*tuple*, **typena**
constexpr tuple (*U* &&*v*, *Us*&&... *vs*)

tuple (*tuple* **const**&)

tuple (*tuple*&&)

template<typename **UTuple**, typename **Enable** = **typename** *std*::enable_if<!*std*::is_same<*tuple*, **typename** *std*::d
constexpr tuple (*UTuple* &&*other*)

tuple &**operator=** (tuple **const** &*other*)

tuple &**operator=** (tuple &&*other*)

template<typename **UTuple**>
tuple &**operator=** (*UTuple* &&*other*)

void **swap** (tuple &*other*)

template<*std*::size_t **I**>
util::at_index<*I*, Ts...>::type &**get** ()

template<*std*::size_t **I**>
util::at_index<*I*, Ts...>::type **const** &**get** () **const**

Private Types

template<>
using index_pack = **typename** *util*::make_index_pack<sizeof...(Ts)>::type

Private Functions

template<*std*::size_t... **Is**, typename **UTuple**>
constexpr tuple (*util*::*index_pack*<*Is*...>, *UTuple* &&*other*)

template<*std*::size_t... **Is**>
void **assign_** (*util*::*index_pack*<*Is*...>, tuple **const** &*other*)

template<*std*::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_for*<Ts...> **_members**

template<>
class tuple<>
#include <tuple.hpp>

Public Functions

constexpr tuple ()

constexpr tuple (*tuple* **const**&)

constexpr 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*>>
#include <tuple.hpp>

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*>>
#include <tuple.hpp>

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*>>
#include <tuple.hpp>

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*)

template<*std*::size_t **I**, typename ...**Ts**>
struct tuple_element<*I*, tuple<*Ts*...>>
#include <tuple.hpp>

Public Types

template<>
using type = **typename** *util*::at_index::type

Public Static Functions

static constexpr type &**get** (*tuple*<Ts...> &*tuple*)

static constexpr type **const** &**get** (*tuple*<Ts...> **const** &*tuple*)

template<class **T**>
struct tuple_size
#include <tuple.hpp> Subclassed by hpx::util::tuple_size< const T >, hpx::util::tuple_size< const volatile T >, hpx::util::tuple_size< volatile T >

namespace adl_barrier

Functions

template<*std*::size_t **I**, typename **Tuple**, typename **Enable** = **typename** *util*::always_void<**typename** *util*::tuple_
constexpr *util*::tuple_element<*I*, *Tuple*>::type &**get** (*Tuple* &*t*)

template<*std*::size_t **I**, typename **Tuple**, typename **Enable** = **typename** *util*::always_void<**typename** *util*::tuple_
constexpr *util*::tuple_element<*I*, *Tuple*>::type **const** &**get** (*Tuple* **const** &*t*)

template<*std*::size_t **I**, typename **Tuple**, typename **Enable** = **typename** *util*::always_void<**typename** *util*::tuple_
constexpr *util*::tuple_element<*I*, *Tuple*>::type &&**get** (*Tuple* &&*t*)

template<*std*::size_t **I**, typename **Tuple**, typename **Enable** = **typename** *util*::always_void<**typename** *util*::tuple_
constexpr *util*::tuple_element<*I*, *Tuple*>::type **const** &&**get** (*Tuple* **const** &&*t*)

namespace std_adl_barrier

Functions

template<*std*::size_t **I**, typename ...**Ts**>
constexpr *util*::tuple_element<*I*, *util*::tuple<*Ts*...>>::type &**get** (*util*::tuple<*Ts*...> &*t*)

template<*std*::size_t **I**, typename ...**Ts**>
constexpr *util*::tuple_element<*I*, *util*::tuple<*Ts*...>>::type **const** &**get** (*util*::tuple<*Ts*...> **const** &*t*)

template<*std*::size_t **I**, typename ...**Ts**>
constexpr *util*::tuple_element<*I*, *util*::tuple<*Ts*...>>::type &&**get** (*util*::tuple<*Ts*...> &&*t*)

template<*std*::size_t **I**, typename ...**Ts**>
constexpr *util*::tuple_element<*I*, *util*::tuple<*Ts*...>>::type **const** &&**get** (*util*::tuple<*Ts*...> **const** &&*t*)

Header `hpx/datastructures/variant_helper.hpp`

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.

Header `hpx/debugging/attach_debugger.hpp`

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.

Header `hpx/debugging/backtrace.hpp`

namespace hpx

namespace util

Functions

std::string **trace**(*std*::size_t *frames_no* = HPX_HAVE_THREAD_BACKTRACE_DEPTH)

Header `hpx/debugging/backtrace/backtrace.hpp`

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
#include <backtrace.hpp>

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*)

Header `hpx/debugging/demangle_helper.hpp`

namespace hpx

namespace util

namespace debug

Typedefs

using cxxabi_demangle_helper = demangle_helper<T>

using cxx_type_id = type_id<T>

Functions

template<typename **T** = void>
std::string **print_type**(**const** char **delim* = "")

template<>
std::string **print_type**(**const** char*)

template<typename **T**, typename ...**Args**>
std::enable_if<sizeof...(*Args*) != 0, *std*::string>::type **print_type**(**const** char **delim* = "")

template<typename **T**>
struct demangle_helper
#include <demangle_helper.hpp>

Public Functions

char **const** ***type_id**() **const**

template<typename **T**>
struct type_id
#include <demangle_helper.hpp>

Public Static Attributes

demangle_helper<T> **typeid_** = demangle_helper<T>()

Header hpx/debugging/print.hpp

Defines

HPX_DP_LAZY(*Expr*, *printer*)

Variables

char ****environ**

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.

Header `hpx/errors/error.hpp`

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

Header `hpx/errors/error_code.hpp`

namespace hpx

Unnamed Group

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

boost::system::error_category **const** &**get_hpx_category** ()
Returns generic HPX error category used for new errors.

boost::system::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*, *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.
- `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*, 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.
- `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*, char **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*, char **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).

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

error_code(int *err*, *hpx*::*exception* **const** &*e*)

error_code(*std*::exception_ptr **const** &*e*)

Private Members

std::exception_ptr **exception_**

Friends

friend hpx::exception

error_code **make_error_code**(*std*::exception_ptr **const** &*e*)

Header hpx/errors/exception.hpp

namespace hpx

Typedefs

using custom_exception_info_handler_type = *std*::function<*hpx*::*exception_info*(*std*::string **const**&, *std*::string **const**&, long, *std*::string **const**&)>

using pre_exception_handler_type = *std*::function<void()>

Functions

void **set_custom_exception_info_handler**(*custom_exception_info_handler_type f*)

void **set_pre_exception_handler**(*pre_exception_handler_type f*)

std::string **get_error_what**(*exception_info* **const** &*xi*)

Return the error message of the thrown exception.

The function hpx::get_error_what can be used to extract the diagnostic information element representing the error message as stored in the given exception instance.

Return The error message stored in the exception 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() *hpx::get_error_backtrace()*, *hpx::get_error_env()*, *hpx::get_error_config()*, *hpx::get_error_state()*

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: hpx::exception_info, hpx::error_code, *std::exception*, or *std::exception_ptr*.

Exceptions

- `std::bad_alloc`: (if one of the required allocations fails)

error **get_error**(*hpx*::*exception* **const** &*e*)

Return the error code value of the exception thrown.

The function hpx::get_error can be used to extract the diagnostic information element representing the error value code as stored in the given exception instance.

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*)

Return the error code value of the exception thrown.

The function hpx::get_error can be used to extract the diagnostic information element representing the error value code as stored in the given exception instance.

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`:

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(), *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)

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(), *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)

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(), *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`:

class exception : public system_error

#include <exception.hpp> A *hpx::exception* is the main exception type used by HPX to report errors.

The *hpx::exception* type is the main exception type used by HPX to report errors. Any exceptions thrown by functions in the HPX library are either of this type or of a type derived from it. This implies that it is always safe to use this type only in catch statements guarding HPX library calls.

Subclassed by *hpx::exception_list*

Public Functions

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(*boost*::system::system_error **const** &*e*)

Construct a *hpx::exception* from a boost::system_error.

exception(*boost*::system::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*

#include <exception.hpp> A *hpx::thread_interrupted* is the exception type used by HPX to interrupt a running HPX thread.

The hpx::thread_interrupted type is the exception type used by HPX to interrupt a running thread.

A running thread can be interrupted by invoking the interrupt() member function of the corresponding hpx::thread object. When the interrupted thread next executes one of the specified interruption points (or if it is currently blocked whilst executing one) with interruption enabled, then a *hpx::thread_interrupted* exception will be thrown in the interrupted thread. If not caught, this will cause the execution of the interrupted thread to terminate. As with any other exception, the stack will be unwound, and destructors for objects of automatic storage duration will be executed.

If a thread wishes to avoid being interrupted, it can create an instance of *hpx::this_thread::disable_interruption*. Objects of this class disable interruption for the thread that created them on construction, and restore the interruption state to whatever it was before on destruction.

```
void f()
{
    // interruption enabled here
    {
        hpx::this_thread::disable_interruption di;
        // interruption disabled
        {
            hpx::this_thread::disable_interruption di2;
            // interruption still disabled
        } // di2 destroyed, interruption state restored
        // interruption still disabled
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

The effects of an instance of *hpx::this_thread::disable_interruption* can be temporarily reversed by constructing an instance of *hpx::this_thread::restore_interruption*, passing in the *hpx::this_thread::disable_interruption* object in question. This will restore the interruption state to what it was when the *hpx::this_thread::disable_interruption* object was constructed, and then disable interruption again when the *hpx::this_thread::restore_interruption* object is destroyed.

```
void g()
{
    // interruption enabled here
    {
        hpx::this_thread::disable_interruption di;
        // interruption disabled
        {
            hpx::this_thread::restore_interruption ri(di);
```

(continues on next page)

(continued from previous page)

```
            // interruption now enabled
        } // ri destroyed, interruption disable again
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

At any point, the interruption state for the current thread can be queried by calling *hpx::this_thread::interruption_enabled()*.

Header `hpx/errors/exception_fwd.hpp`

namespace hpx

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.

Header hpx/errors/exception_info.hpp

Defines

HPX_DEFINE_ERROR_INFO(*NAME*, *TYPE*)

namespace hpx

Functions

template<typename E>HPX_NORETURN void hpx::throw_with_info(E && e, exception_info && i

template<typename E>HPX_NORETURN void hpx::throw_with_info(E && e, exception_info cons

template<typename **E**>
exception_info ***get_exception_info**(*E &e*)

template<typename **E**>
exception_info **const** ***get_exception_info**(*E* **const** *&e*)

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*)

template<typename **Tag**, typename **Type**>
struct error_info
#include <exception_info.hpp>

Public Types

template<>
using tag = Tag

template<>
using type = Type

Public Functions

error_info(Type **const** *&value*)

error_info(Type *&&value*)

Public Members

Type **_value**

class exception_info

#include <exception_info.hpp> 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**

Header hpx/errors/exception_list.hpp

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

typedef *exception_list_type*::const_iterator **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*.

Private Types

typedef *boost*::detail::spinlock **mutex_type**

typedef *std*::list<*std*::exception_ptr> **exception_list_type**

Private Members

exception_list_type **exceptions_**

mutex_type **mtx_**

Header hpx/errors/throw_exception.hpp

Defines

HPX_THROW_EXCEPTION(*errcode*, *f*, *msg*)
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*, *msg*)

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.

Header `hpx/exception.hpp`

Header `hpx/system_error.hpp`

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.

Header `hpx/execution/execution.hpp`

Header `hpx/execution/executor_parameters.hpp`

Header `hpx/execution/executors/auto_chunk_size.hpp`

namespace hpx

namespace parallel

namespace execution

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 (*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.

auto_chunk_size (*hpx*::*util*::*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.

Header `hpx/execution/executors/dynamic_chunk_size.hpp`

namespace hpx

namespace parallel

namespace execution

struct dynamic_chunk_size
#include <dynamic_chunk_size.hpp> Loop iterations are divided into pieces of size *chunk_size* and then dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another If *chunk_size* is not specified, the default chunk size is 1.

Note This executor parameters type is equivalent to OpenMP's DYNAMIC scheduling directive.

Public Functions

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.

Header `hpx/execution/executors/execution.hpp`

Header `hpx/execution/executors/execution_information.hpp`

Header `hpx/execution/executors/execution_information_fwd.hpp`

Header `hpx/execution/executors/execution_parameters.hpp`

namespace hpx

namespace parallel

namespace execution

Functions

template<typename ...**Params**>
executor_parameters_join<Params...>::type **join_executor_parameters**(*Params&&... params*)

template<typename **Param**>
Param &&**join_executor_parameters**(*Param* &&*param*)

template<typename ...**Params**>
struct executor_parameters_join
#include <execution_parameters.hpp>

Public Types

template<>
using type = detail::executor_parameters<**typename** *hpx*::*util*::decay<Params>::type...>

template<typename **Param**>
struct executor_parameters_join<*Param*>
#include <execution_parameters.hpp>

Public Types

template<>
using type = Param

Header `hpx/execution/executors/execution_parameters_fwd.hpp`

Header `hpx/execution/executors/fused_bulk_execute.hpp`

Header `hpx/execution/executors/guided_chunk_size.hpp`

namespace hpx

namespace parallel

namespace execution

struct guided_chunk_size
#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

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.

Header `hpx/execution/executors/persistent_auto_chunk_size.hpp`

namespace hpx

namespace parallel

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

constexpr persistent_auto_chunk_size (*std*::uint64_t *num_iters_for_timing* = 0)

Construct an persistent_auto_chunk_size executor parameters object

Note Default constructed persistent_auto_chunk_size executor parameter types will use 0 microseconds as the execution time for each chunk and 80 microseconds as the minimal time for which any of the scheduled chunks should run.

persistent_auto_chunk_size (*hpx*::*util*::*steady_duration* **const** &*time_cs*, *std*::uint64_t *num_iters_for_timing* = 0)

Construct an persistent_auto_chunk_size executor parameters object

Parameters

- `time_cs`: The execution time for each chunk.

persistent_auto_chunk_size (*hpx*::*util*::*steady_duration* **const** &*time_cs*, *hpx*::*util*::*steady_duration* **const** &*rel_time*, *std*::uint64_t *num_iters_for_timing* = 0)

Construct an persistent_auto_chunk_size executor parameters object

Parameters

- `rel_time`: [in] The time duration to use as the minimum to decide how many loop iterations should be combined.
- `time_cs`: The execution time for each chunk.

Header hpx/execution/executors/polymorphic_executor.hpp

namespace hpx

namespace parallel

namespace execution

template<typename **R**, typename ...**Ts**>
class polymorphic_executor<*R* (*Ts*...)> : **private** *hpx*::*parallel*::*execution*::detail::polymorphic_executor_base

#include <polymorphic_executor.hpp>

Public Types

template<>
using future_type = *hpx*::future<R>

Public Functions

constexpr 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 &**operator=** (*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**()

Header `hpx/execution/executors/rebind_executor.hpp`

namespace hpx

namespace parallel

namespace execution

template<typename **ExecutionPolicy**, 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

typedef ExecutionPolicy::template rebind<executor_type, parameters_type>::type **type**
The type of the rebound execution policy.

Header `hpx/execution/executors/static_chunk_size.hpp`

namespace hpx

namespace parallel

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.

Header `hpx/execution/traits/executor_traits.hpp`

namespace hpx

namespace parallel

namespace execution

Typedefs

using executor_future_t = typename executor_future<Executor, T, Ts...>::type

template<typename **Executor**>
struct executor_context
#include <executor_traits.hpp>

Public Types

template<>
using type = typename *std*::decay::type

template<typename **Executor**>
struct executor_execution_category
#include <executor_traits.hpp>

Public Types

template<>
using type = *hpx*::*util*::*detected_or_t*<*unsequenced_execution_tag*, *execution_category*, Executor>

Private Types

template<>
using execution_category = typename T::execution_category

template<typename **Executor**>
struct executor_index
#include <executor_traits.hpp>

Public Types

template<>
using type = *hpx*::*util*::*detected_or_t*<**typename** executor_shape<Executor>::*type*, *index_type*, Executor>

Private Types

template<>
using index_type = typename T::index_type

template<typename **Executor**>
struct executor_parameters_type
#include <executor_traits.hpp>

Public Types

template<>
using type = *hpx*::*util*::*detected_or_t*<*parallel*::*execution*::*static_chunk_size*, *parameters_type*, Executor>

Private Types

template<>
using parameters_type = **typename** T::parameters_type

template<typename **Executor**>
struct executor_shape
#include <executor_traits.hpp>

Public Types

template<>
using type = *hpx*::*util*::*detected_or_t*<*std*::size_t, *shape_type*, Executor>

Private Types

template<>
using shape_type = **typename** T::shape_type

namespace traits

Typedefs

using executor_context_t = **typename** executor_context<Executor>::type

using executor_execution_category_t = **typename** executor_execution_category<Executor>::type

using executor_shape_t = **typename** executor_shape<Executor>::type

using executor_index_t = **typename** executor_index<Executor>::type

using executor_future_t = **typename** executor_future<Executor, T, Ts...>::type

using executor_parameters_type_t = **typename** executor_parameters_type<Executor>::type

Header `hpx/execution/traits/future_then_result_exec.hpp`

Header `hpx/execution/traits/is_execution_policy.hpp`

namespace hpx

namespace parallel

namespace execution

template<typename **T**>

struct is_async_execution_policy : **public** *execution*::detail::is_async_execution_policy<*hpx*::*util*::dec

#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** *execution*::detail::is_execution_policy<*hpx*::*util*::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 function signatures from otherwise 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** *execution*::detail::is_parallel_execution_policy<*hpx*::*ut*

#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** *execution*::detail::is_sequenced_execution_policy<*hp*

#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.

Header hpx/execution/traits/is_executor.hpp

namespace hpx

namespace parallel

namespace execution

Typedefs

using is_one_way_executor_t = **typename** is_one_way_executor<T>::type

using is_never_blocking_one_way_executor_t = **typename** is_never_blocking_one_way_executor<T

using is_bulk_one_way_executor_t = **typename** is_bulk_one_way_executor<T>::type

using is_two_way_executor_t = **typename** is_two_way_executor<T>::type

using is_bulk_two_way_executor_t = **typename** is_bulk_two_way_executor<T>::type

namespace traits

Typedefs

using is_one_way_executor_t = **typename** is_one_way_executor<T>::type

using is_never_blocking_one_way_executor_t = **typename** is_never_blocking_one_way_executor<T>::ty

using is_bulk_one_way_executor_t = **typename** is_bulk_one_way_executor<T>::type

using is_two_way_executor_t = **typename** is_two_way_executor<T>::type

using is_bulk_two_way_executor_t = **typename** is_bulk_two_way_executor<T>::type

using is_executor_any_t = **typename** is_executor_any<T>::type

Header hpx/execution/traits/is_executor_parameters.hpp

template<typename **Executor**>
struct extract_executor_parameters<*Executor*, **typename** *hpx*::*util*::always_void<**typename** *Executor*::executor_para
#include <is_executor_parameters.hpp>

Public Types

template<>
using type = **typename** Executor::executor_parameters_type

template<typename **Parameters**>
struct extract_has_variable_chunk_size<*Parameters*, **typename** *hpx*::*util*::always_void<**typename** *Parameters*::has
#include <is_executor_parameters.hpp>

Public Types

template<>
using type = **typename** Parameters::has_variable_chunk_size

namespace hpx

namespace parallel

namespace execution

Typedefs

using is_executor_parameters_t = **typename** is_executor_parameters<T>::type

template<typename **Executor**, typename **Enable** = void>
struct extract_executor_parameters
#include <is_executor_parameters.hpp>

Public Types

template<>
using type = sequential_executor_parameters

template<typename **Executor**>
struct extract_executor_parameters<*Executor*, **typename** *hpx*::*util*::always_void<**typename** *Executo*
#include <is_executor_parameters.hpp>

Public Types

template<>
using type = **typename** Executor::executor_parameters_type

template<typename **Parameters**, typename **Enable** = void>
struct extract_has_variable_chunk_size
#include <is_executor_parameters.hpp>

Public Types

template<>
using type = *std*::false_type

template<typename **Parameters**>
struct extract_has_variable_chunk_size<*Parameters*, **typename** *hpx*::*util*::always_void<**typename**
#include <is_executor_parameters.hpp>

Public Types

template<>
using type = **typename** Parameters::has_variable_chunk_size

namespace traits

Typedefs

using is_executor_parameters_t = **typename** is_executor_parameters<T>::type

Header `hpx/execution/traits/vector_pack_alignment_size.hpp`

Header `hpx/execution/traits/vector_pack_count_bits.hpp`

namespace hpx

namespace parallel

namespace traits

Functions

std::size_t **count_bits** (bool *value*)

Header `hpx/execution/traits/vector_pack_load_store.hpp`

Header `hpx/execution/traits/vector_pack_type.hpp`

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.

Header `hpx/execution_base/agent_base.hpp`

namespace hpx

namespace execution_base

struct agent_base
#include <agent_base.hpp>

Public Functions

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*::*util*::*steady_duration* **const** &*sleep_duration*, char **const** **desc*) = 0

virtual void **sleep_until**(*hpx*::*util*::*steady_time_point* **const** &*sleep_time*, char **const** **desc*) = 0

Header `hpx/execution_base/agent_ref.hpp`

namespace hpx

namespace execution_base

class agent_ref

#include <agent_ref.hpp>

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(*agent_ref*&&)

constexpr *agent_ref* &**operator=**(*agent_ref*&&)

constexpr operator bool() const

void **reset**(*agent_base* **impl* = nullptr)

void **yield**(char **const** **desc* = "hpx::execution_base::agent_ref::yield")

void **yield_k**(*std*::size_t *k*, char **const** **desc* = "hpx::execution_base::agent_ref::yield_k")

void **suspend**(char **const** **desc* = "hpx::execution_base::agent_ref::suspend") void **resume**(char **const** *desc = "hpx::execution_base::agent_ref::resume")

void **abort**(char **const** *desc = "hpx::execution_base::agent_ref::abort")

template<typename **Rep**, typename **Period**>
void **sleep_for**(*std*::chrono::duration<*Rep*, *Period*> **const** &*sleep_duration*, char **const** *desc = "hpx::execution_base::agent_ref::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_ref::sleep_until")

agent_base &**ref**()

Private Functions

void **sleep_for**(*hpx*::*util*::*steady_duration* **const** &*sleep_duration*, char **const** *desc)

void **sleep_until**(*hpx*::*util*::*steady_time_point* **const** &*sleep_time*, char **const** *desc)

Private Members

agent_base ***impl_**

Friends

constexpr bool **operator==**(agent_ref **const** &*lhs*, agent_ref **const** &*rhs*)

constexpr bool **operator!=**(agent_ref **const** &*lhs*, agent_ref **const** &*rhs*)

std::ostream &**operator<<**(*std*::ostream&, agent_ref **const**&)

Header `hpx/execution_base/context_base.hpp`

namespace hpx

namespace execution_base

struct context_base
#include <context_base.hpp>

Public Functions

virtual ~context_base()

virtual *resource_base* **const** &**resource**() **const** = 0

Header `hpx/execution_base/execution.hpp`

namespace hpx

namespace parallel

namespace execution

struct parallel_execution_tag

#include <execution.hpp> Function invocations executed by a group of parallel execution agents execute in unordered fashion. Any such invocations executing in the same thread are indeterminately sequenced with respect to each other.

Note parallel_execution_tag is weaker than sequenced_execution_tag.

struct sequenced_execution_tag

#include <execution.hpp> Function invocations executed by a group of sequential execution agents execute in sequential order.

struct unsequenced_execution_tag

#include <execution.hpp> Function invocations executed by a group of vector execution agents are permitted to execute in unordered fashion when executed in different threads, and unsequenced with respect to one another when executed in the same thread.

Note unsequenced_execution_tag is weaker than parallel_execution_tag.

Header `hpx/execution_base/register_locks.hpp`

namespace hpx

namespace util

Functions

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**()

constexpr void **set_held_locks_data**(*std*::unique_ptr<held_locks_data>&&)

struct ignore_all_while_checking
#include <register_locks.hpp>

Public Functions

ignore_all_while_checking()

template<typename **Lock**, typename **Enable**>
struct ignore_while_checking
#include <register_locks.hpp>

Public Functions

ignore_while_checking(void **const***)

Header `hpx/execution_base/resource_base.hpp`

namespace hpx

namespace execution_base

struct resource_base
#include <resource_base.hpp> TODO: implement, this is currently just a dummy.

Public Functions

virtual ~resource_base()

Header `hpx/execution_base/this_thread.hpp`

namespace hpx

namespace execution_base

namespace this_thread

Functions

hpx::*execution_base*::*agent_ref* **agent**()

void **yield**(char **const** **desc* = "hpx::execution_base::this_thread::yield")

void **yield_k**(*std*::size_t *k*, char **const** **desc* = "hpx::execution_base::this_thread::yield_k")

void **suspend**(char **const** **desc* = "hpx::execution_base::this_thread::suspend")

template<typename **Rep**, typename **Period**>
void **sleep_for**(*std*::chrono::duration<*Rep*, *Period*> **const** &*sleep_duration*, char **const** **desc* = "hpx::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* = "hpx::execution_base::this_thread::sleep_for")

struct reset_agent
#include <this_thread.hpp>

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)

executors

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.

Header `hpx/executors/apply.hpp`

Header `hpx/executors/async.hpp`

Header `hpx/executors/current_executor.hpp`

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 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 threads

Functions

parallel::execution::current_executor **get_executor**(*thread_id_type* **const** *&id*, *error_code &ec = throws*)

Returns a reference to the executor which was used to create the given 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*.

Header `hpx/executors/dataflow.hpp`

Header `hpx/executors/datapar/execution_policy.hpp`

Header `hpx/executors/datapar/execution_policy_fwd.hpp`

Header `hpx/executors/exception_list.hpp`

Header `hpx/executors/execution_policy.hpp`

namespace hpx

namespace parallel

namespace execution

Variables

constexpr task_policy_tag **task**
Default sequential execution policy object.

constexpr *sequenced_policy* **seq**
Default sequential execution policy object.

constexpr *parallel_policy* **par**
Default parallel execution policy object.

constexpr *parallel_unsequenced_policy* **par_unseq**
Default vector execution policy object.

struct parallel_policy
#include <execution_policy.hpp> The class *parallel_policy* is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

Subclassed by *hpx::parallel::execution::parallel_policy_shim< Executor, Parameters >*

Public Types

typedef *parallel_executor* **executor_type**
The type of the executor associated with this execution policy.

typedef *execution*::*extract_executor_parameters*<*executor_type*>::type **executor_parameters_type**
The type of the associated executor parameters object which is associated with this execution policy

typedef *parallel_execution_tag* **execution_category**
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

template<typename **Executor**>
rebind_executor<*parallel_policy*, *Executor*, *executor_parameters_type*>::type **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**, typename **ParametersType** = **typename** *executor_parameters_join<Pa*
rebind_executor<*parallel_policy*, *executor_type*, *ParametersType*>::type **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**>
void **serialize**(*Archive* &*ar*, **const** unsigned int *version*)

Private Members

executor_type **exec_**

executor_parameters_type **params_**

Friends

friend hpx::parallel::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

typedef parallel_policy_shim<Executor_, Parameters_> **type**
The type of the rebound execution policy.

template<typename **Executor**, typename **Parameters**>
struct parallel_policy_shim : **public** *hpx*::*parallel*::*execution*::*parallel_policy*
#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

typedef Executor **executor_type**
The type of the executor associated with this execution policy.

typedef Parameters **executor_parameters_type**
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 parallel_task_policy_shim<Executor, Parameters> **operator()**(task_policy_tag *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

template<typename **Executor_**>
rebind_executor<parallel_policy_shim, *Executor_*, *executor_parameters_type*>::type **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_**, typename **ParametersType** = **typename** *executor_parameters_join<I*
rebind_executor<parallel_policy_shim, *executor_type*, *ParametersType*>::type **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 &**executor**()

Return the associated executor object.

constexpr Executor **const** &**executor**() **const**

Return the associated executor object.

Parameters &**parameters**()

Return the associated executor parameters object.

constexpr Parameters **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<>
typedef parallel_policy_shim<Executor_, Parameters_> **type**

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*.

Subclassed by *hpx::parallel::execution::parallel_task_policy_shim< Executor, Parameters >*

Public Types

typedef *parallel_executor* **executor_type**
The type of the executor associated with this execution policy.

typedef *execution::extract_executor_parameters<executor_type>*::type **executor_parameters_type**
The type of the associated executor parameters object which is associated with this execution policy

typedef *parallel_execution_tag* **execution_category**
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

template<typename **Executor**>
rebind_executor<*parallel_task_policy*, *Executor*, *executor_parameters_type*>::type **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**, typename **ParametersType** = **typename** *executor_parameters_join<Pa*
rebind_executor<*parallel_task_policy*, *executor_type*, *ParametersType*>::type **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.

Private Functions

template<typename **Archive**>
void **serialize**(*Archive* &*ar*, **const** unsigned int *version*)

Private Members

executor_type **exec_**

executor_parameters_type **params_**

Friends

friend hpx::parallel::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

typedef parallel_task_policy_shim<Executor_, Parameters_> **type**
The type of the rebound execution policy.

template<typename **Executor**, typename **Parameters**>
struct parallel_task_policy_shim : **public** *hpx*::*parallel*::*execution*::*parallel_task_policy*
#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

typedef Executor **executor_type**
The type of the executor associated with this execution policy.

typedef Parameters **executor_parameters_type**
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 parallel_task_policy_shim **operator()** (task_policy_tag *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

template<typename **Executor_**>
rebind_executor<parallel_task_policy_shim, *Executor_*, *executor_parameters_type*>::type **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_**, typename **ParametersType** = **typename** *executor_parameters_join<I*
rebind_executor<parallel_task_policy_shim, *executor_type*, *ParametersType*>::type **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 &**executor** ()

Return the associated executor object.

constexpr Executor **const** &**executor** () **const**

Return the associated executor object.

Parameters &**parameters** ()

Return the associated executor parameters object.

constexpr Parameters **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<>
typedef parallel_task_policy_shim<Executor_, Parameters_> **type**
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 vectorized.

Public Types

typedef *parallel_executor* **executor_type**
The type of the executor associated with this execution policy.

typedef *execution*::*extract_executor_parameters*<*executor_type*>::type **executor_parameters_type**
The type of the associated executor parameters object which is associated with this execution policy

typedef *parallel_execution_tag* **execution_category**
The category of the execution agents created by this execution policy.

Public Functions

parallel_unsequenced_policy **operator()** (task_policy_tag) **const**
Create a new *parallel_unsequenced_policy* from itself

Return The new *parallel_unsequenced_policy*
Parameters
- `tag`: [in] Specify that the corresponding asynchronous execution policy should be used

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**>
void **serialize** (*Archive* &*ar*, **const** unsigned int *version*)

Private Members

executor_type **exec_**

executor_parameters_type **params_**

Friends

friend hpx::parallel::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.

Subclassed by *hpx::parallel::execution::sequenced_policy_shim< Executor, Parameters >*

Public Types

typedef *sequenced_executor* **executor_type**

The type of the executor associated with this execution policy.

typedef *execution::extract_executor_parameters<executor_type>*::type **executor_parameters_type**

The type of the associated executor parameters object which is associated with this execution policy

typedef *sequenced_execution_tag* **execution_category**

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

template<typename **Executor**>
rebind_executor<*sequenced_policy*, *Executor*, *executor_parameters_type*>::type **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**, typename **ParametersType** = **typename** *executor_parameters_join<Pa*

rebind_executor<*sequenced_policy*, *executor_type*, *ParametersType*>::type **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.

executor_type &**executor**()

Return the associated executor object. 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**>
void **serialize**(*Archive* &*ar*, **const** unsigned int *version*)

Private Members

executor_type **exec_**

executor_parameters_type **params_**

Friends

friend hpx::parallel::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

typedef sequenced_policy_shim<Executor_, Parameters_> **type**
The type of the rebound execution policy.

template<typename **Executor**, typename **Parameters**>
struct sequenced_policy_shim : **public** *hpx*::*parallel*::*execution*::*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

typedef Executor **executor_type**
The type of the executor associated with this execution policy.

typedef Parameters **executor_parameters_type**
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<Executor, Parameters> **operator()** (task_policy_tag *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

template<typename **Executor_**>
rebind_executor<sequenced_policy_shim, *Executor_*, *executor_parameters_type*>::type **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_**, typename **ParametersType** = **typename** *executor_parameters_join<I*
rebind_executor<sequenced_policy_shim, *executor_type*, *ParametersType*>::type **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 &**executor**()

Return the associated executor object.

constexpr Executor **const** &**executor**() **const**

Return the associated executor object.

Parameters &**parameters**()

Return the associated executor parameters object.

constexpr Parameters **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<>
typedef sequenced_policy_shim<Executor_, Parameters_> **type**

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 *sequenced_policy*.

Subclassed by *hpx::parallel::execution::sequenced_task_policy_shim< Executor, Parameters >*

Public Types

typedef *sequenced_executor* **executor_type**

The type of the executor associated with this execution policy.

typedef *execution*::*extract_executor_parameters*<*executor_type*>::type **executor_parameters_type**

The type of the associated executor parameters object which is associated with this execution policy

typedef *sequenced_execution_tag* **execution_category**

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* from itself

Return The new *sequenced_task_policy*

Parameters

- `tag`: [in] Specify that the corresponding asynchronous execution policy should be used

template<typename **Executor**>
rebind_executor<*sequenced_task_policy*, *Executor*, *executor_parameters_type*>::type **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**, typename **ParametersType** = **typename** *executor_parameters_join<Pa*
rebind_executor<*sequenced_task_policy*, *executor_type*, *ParametersType*>::type **with** (*Parameters&&... params*) **const**

Create a new *sequenced_task_policy* 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*

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**>
void **serialize**(*Archive &ar*, **const** unsigned int *version*)

Private Members

executor_type **exec_**

executor_parameters_type **params_**

Friends

friend hpx::parallel::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

typedef sequenced_task_policy_shim<Executor_, Parameters_> **type**

The type of the rebound execution policy.

template<typename **Executor**, typename **Parameters**>
struct sequenced_task_policy_shim : **public** *hpx*::*parallel*::*execution*::*sequenced_task_policy*

#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

typedef Executor **executor_type**

The type of the executor associated with this execution policy.

typedef Parameters **executor_parameters_type**

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 *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

template<typename **Executor_**>
rebind_executor<sequenced_task_policy_shim, *Executor_*, *executor_parameters_type*>::type **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_**, typename **ParametersType** = **typename** *executor_parameters_join<I*
rebind_executor<sequenced_task_policy_shim, *executor_type*, *ParametersType*>::type **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 &**executor**()

Return the associated executor object.

constexpr Executor **const** &**executor**() **const**

Return the associated executor object.

Parameters &**parameters**()

Return the associated executor parameters object.

constexpr Parameters **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<>
typedef sequenced_task_policy_shim<Executor_, Parameters_> **type**
The type of the rebound execution policy.

Header `hpx/executors/execution_policy_fwd.hpp`

Header `hpx/executors/guided_pool_executor.hpp`

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*>>
#include <guided_pool_executor.hpp>

Public Types

typedef *parallel*::*execution*::*parallel_execution_tag* **type**

template<typename **Hint**>
struct executor_execution_category<guided_pool_executor_shim<*Hint*>>
#include <guided_pool_executor.hpp>

Public Types

typedef *parallel*::*execution*::*parallel_execution_tag* **type**

template<typename **Tag**>
struct guided_pool_executor<pool_numa_hint<*Tag*>>
#include <guided_pool_executor.hpp>

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 = detail::enable_if_t<*hpx*::*traits*::*is_future*<*F*
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*::*util*::tuple<*InnerFutures*...>> &&*predecessor*, *Ts*&&... *ts*)

template<typename **F**, typename ...**InnerFutures**, typename = detail::enable_if_t<*hpx*::*traits*::is_future_tuple<
auto **async_execute**(*F* &&*f*, *hpx*::*util*::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

#include <guided_pool_executor.hpp>

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 = detail::enable_if_t<*hpx*::*traits*::*is_future*<*F*
auto **then_execute**(*F* &&*f*, *Future* &&*predecessor*, *Ts*&&... *ts*)

Public Members

bool **guided_**

guided_pool_executor<H> **guided_exec_**

Header `hpx/executors/limiting_executor.hpp`

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
#include <limiting_executor.hpp>

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**>
decltype(auto) **bulk_async_execute** (*F &&f*, *S* **const** *&shape*, *Ts&&... ts*)

template<typename **F**, typename **S**, typename **Future**, typename ...**Ts**>
decltype(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
#include <limiting_executor.hpp>

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
#include <limiting_executor.hpp>

Public Functions

template<>
throttling_wrapper (limiting_executor &*lim*, BaseExecutor **const** &*base*, F &&*f*)

template<typename ...**Ts**>
decltype(auto) **operator()** (Ts&&... *ts*)

template<>
bool **exceeds_upper** ()

template<>
bool **exceeds_lower** ()

Public Members

template<>
limiting_executor &**limiting_**

template<>
F **f_**

Header hpx/executors/parallel_executor.hpp

namespace hpx

namespace parallel

namespace execution

Typedefs

using parallel_executor = parallel_policy_executor<*hpx*::*launch*>

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

typedef *parallel_execution_tag* **execution_category**
Associate the parallel_execution_tag executor tag type as a default with this executor.

typedef *static_chunk_size* **executor_parameters_type**
Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

constexpr parallel_policy_executor(*threads*::*thread_priority* *priority* = *threads*::*thread_priority_default*, *threads*::*thread_stacksize* *stacksize* = *threads*::*thread_stacksize_default*, *threads*::*thread_schedule_hint* *schedulehint* = {}, Policy *l* = detail::get_default_policy<Policy>::call(), *std*::size_t *spread* = 4, *std*::size_t *tasks* = *std*::size_t(-1))

Create a new parallel executor.

constexpr parallel_policy_executor(*threads*::*thread_stacksize stacksize*, *threads*::*thread_schedule_hint schedulehint* = {}, Policy *l* = detail::get_default_policy<Policy>::call(), *std*::size_t *spread* = 4, *std*::size_t *tasks* = *std*::size_t(-1))

constexpr parallel_policy_executor(*threads*::*thread_schedule_hint schedulehint*, Policy *l* = detail::get_default_policy<Policy>::call(), *std*::size_t *spread* = 4, *std*::size_t *tasks* = *std*::size_t(-1))

constexpr parallel_policy_executor(Policy *l*, *std*::size_t *spread* = 4, *std*::size_t *tasks* = *std*::size_t(-1))

Header `hpx/executors/parallel_executor_aggregated.hpp`

template<>
struct parallel_policy_executor_aggregated<*hpx*::*launch*>
#include <parallel_executor_aggregated.hpp>

Public Types

template<>
using execution_category = *parallel_execution_tag*
Associate the parallel_execution_tag executor tag type as a default with this executor.

template<>
using executor_parameters_type = *static_chunk_size*
Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

constexpr 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*) **const**

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

template<>
using execution_category = *parallel_execution_tag*

Associate the parallel_execution_tag executor tag type as a default with this executor.

template<>
using executor_parameters_type = *static_chunk_size*

Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

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**

template<>
struct parallel_policy_executor_aggregated<*hpx*::*launch*>

#include <parallel_executor_aggregated.hpp>

Public Types

template<>
using execution_category = *parallel_execution_tag*

Associate the parallel_execution_tag executor tag type as a default with this executor.

template<>
using executor_parameters_type = *static_chunk_size*

Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

constexpr 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*) **const**

Header `hpx/executors/restricted_thread_pool_executor.hpp`

namespace hpx

namespace parallel

namespace execution

class restricted_thread_pool_executor
#include <restricted_thread_pool_executor.hpp>

Public Types

typedef *parallel_execution_tag* **execution_category**
Associate the parallel_execution_tag executor tag type as a default with this executor.

typedef *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* *schedulehint* = {}, *std*::size_t *hierarchical_threshold* = *hierarchical_threshold_default_*)

Create a new parallel executor.

restricted_thread_pool_executor(*restricted_thread_pool_executor* **const** &*other*)

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* **schedulehint_** = {}

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

Header `hpx/executors/sequenced_executor.hpp`

namespace hpx

namespace parallel

namespace execution

struct sequenced_executor

#include <sequenced_executor.hpp> A *sequential_executor* creates groups of sequential execution agents which execute in the calling thread. The sequential order is given by the lexicographical order of indices in the index space.

Header `hpx/executors/service_executors.hpp`

Header `hpx/executors/sync.hpp`

Header `hpx/executors/thread_pool_attached_executors.hpp`

Header `hpx/executors/thread_pool_executor.hpp`

namespace hpx

namespace parallel

namespace execution

class thread_pool_executor

#include <thread_pool_executor.hpp> A thread_pool_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

typedef *parallel_execution_tag* **execution_category**

Associate the parallel_execution_tag executor tag type as a default with this executor.

typedef *static_chunk_size* **executor_parameters_type**

Associate the static_chunk_size executor parameters type as a default with this executor.

Public Functions

thread_pool_executor(*threads*::*thread_priority priority* = *threads*::*thread_priority_default*, *threads*::*thread_stacksize stacksize* = *threads*::*thread_stacksize_default*, *threads*::*thread_schedule_hint schedulehint* = {}, *std*::size_t *hierarchical_threshold* = *hierarchical_threshold_default_*)

Create a new parallel executor.

thread_pool_executor(*threads*::*policies*::*scheduler_base* **scheduler*, *threads*::*thread_priority priority* = *threads*::*thread_priority_default*, *threads*::*thread_stacksize stacksize* = *threads*::*thread_stacksize_default*, *threads*::*thread_schedule_hint schedulehint* = {}, *std*::size_t *hierarchical_threshold* = *hierarchical_threshold_default_*)

thread_pool_executor(*threads*::*thread_pool_base* **pool*, *threads*::*thread_priority priority* = *threads*::*thread_priority_default*, *threads*::*thread_stacksize stacksize* = *threads*::*thread_stacksize_default*, *threads*::*thread_schedule_hint schedulehint* = {}, *std*::size_t *hierarchical_threshold* = *hierarchical_threshold_default_*)

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* **schedulehint_** = {}

std::size_t **hierarchical_threshold_** = *hierarchical_threshold_default_*

Private Static Attributes

constexpr *std*::size_t **hierarchical_threshold_default_** = 6

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.

Header `hpx/executors_distributed/distribution_policy_executor.hpp`

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.

Header `hpx/modules/filesystem.hpp`

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

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*, *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.

Header `hpx/format.hpp`

Header `hpx/modules/format.hpp`

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*)

Header `hpx/util/bad_lexical_cast.hpp`

namespace hpx

namespace util

class bad_lexical_cast : **public** bad_cast
#include <bad_lexical_cast.hpp>

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

std::type_info **const** ***source**

std::type_info **const** ***target**

Header `hpx/util/from_string.hpp`

namespace hpx

namespace util

Functions

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*)

Header `hpx/util/to_string.hpp`

namespace hpx

namespace util

Functions

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.

Header hpx/functional/bind.hpp

namespace hpx

namespace serialization

Functions

template<typename **Archive**, typename **F**, typename ...**Ts**>
void **serialize**(*Archive &ar*, ::*hpx*::*util*::detail::bound<*F*, *Ts*...> *&bound*, unsigned int **const** *version* = 0)

template<typename **Archive**, *std*::size_t **I**>
void **serialize**(*Archive &ar*, ::*hpx*::*util*::detail::placeholder<*I*>&, unsigned int **const** = 0)

namespace util

Functions

template<typename **F**, typename ...**Ts**, typename **Enable** = **typename** *std*::enable_if<!*traits*::is_action<**typename** *std*::
constexpr detail::bound<**typename** *std*::decay<*F*>::type, **typename** *util*::make_index_pack<sizeof...(*Ts*)>::type, **typ**

namespace placeholders

Variables

constexpr detail::placeholder<1> **_1** = {}

constexpr detail::placeholder<2> **_2** = {}

constexpr detail::placeholder<3> **_3** = {}

constexpr detail::placeholder<4> **_4** = {}

constexpr detail::placeholder<5> **_5** = {}

constexpr detail::placeholder<6> **_6** = {}

constexpr detail::placeholder<7> **_7** = {}

constexpr detail::placeholder<8> **_8** = {}

constexpr detail::placeholder<9> **_9** = {}

Header hpx/functional/bind_back.hpp

namespace hpx

namespace serialization

Functions

template<typename **Archive**, typename **F**, typename ...**Ts**>
void **serialize**(*Archive* &*ar*, ::*hpx*::*util*::detail::bound_back<*F*, *Ts*...> &*bound*, unsigned int **const** *version* = 0)

namespace util

Functions

template<typename **F**, typename ...**Ts**>
constexpr detail::bound_back<**typename** *std*::decay<*F*>::type, **typename** *util*::make_index_pack<sizeof...(*Ts*)>::typ

template<typename **F**>
constexpr *std*::decay<*F*>::type **bind_back**(*F* &&*f*)

Header hpx/functional/bind_front.hpp

namespace hpx

namespace serialization

Functions

template<typename **Archive**, typename **F**, typename ...**Ts**>
void **serialize**(*Archive* &*ar*, ::*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*)>::typ

template<typename **F**>
constexpr *std*::decay<*F*>::type **bind_front**(*F* &&*f*)

Header hpx/functional/deferred_call.hpp

namespace hpx

namespace serialization

Functions

template<typename **Archive**, typename **F**, typename ...**Ts**>
void **serialize** (*Archive* &*ar*, ::*hpx*::*util*::detail::deferred<*F*, *Ts*...> &*d*, unsigned int **const** *version* = 0)

namespace util

Functions

template<typename **F**, typename ...**Ts**>
detail::deferred<**typename** *std*::decay<*F*>::type, **typename** *util*::make_index_pack<sizeof...(*Ts*)>::type, **typename** *util*

template<typename **F**>
std::decay<*F*>::type **deferred_call** (*F* &&*f*)

Header hpx/functional/first_argument.hpp

Header hpx/functional/function.hpp

Defines

HPX_UTIL_REGISTER_FUNCTION_DECLARATION (*Sig*, *F*, *Name*)

HPX_UTIL_REGISTER_FUNCTION (*Sig*, *F*, *Name*)

namespace hpx

namespace util

Typedefs

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*> *#include <function.hpp>*

Public Types

typedef R **result_type**

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*::enabl
function (*F* &&*f*)

template<typename **F**, typename **FD** = **typename** *std*::decay<*F*>::type, typename **Enable1** = **typename** *std*::enabl
function &**operator=** (*F* &&*f*)

Private Types

template<>
using base_type = detail::basic_function<R (Ts...) , true, Serializable>

Header `hpx/functional/function_ref.hpp`

namespace hpx

namespace util

template<typename **R**, typename ...**Ts**>
class function_ref<*R* (*Ts*...) >
#include <function_ref.hpp>

Public Functions

template<typename **F**, typename **FD** = **typename** *std*::decay<*F*>::type, typename **Enable** = **typename** *std*::enable_
function_ref (*F* &&*f*)

function_ref (*function_ref* **const** &*other*)

template<typename **F**, typename **FD** = **typename** *std*::decay<*F*>::type, typename **Enable** = **typename** *std*::enable_
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** *s*
void **assign** (*F* &&*f*)

template<typename **T**>
void **assign**(*std*::reference_wrapper<*T*> *f_ref*)

template<typename **T**>
void **assign**(*T* **f_ptr*)

void **swap**(function_ref &*f*)

R **operator()**(Ts... *vs*) **const**

std::size_t **get_function_address**() **const**

char **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**()

Header `hpx/functional/invoke.hpp`

Defines

HPX_INVOKE(*F*, ...)

HPX_INVOKE_R(*R*, *F*, ...)

namespace 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*)
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 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
#include <invoke.hpp>

Public Functions

template<typename **F**, typename ...**Ts**>
constexpr *util*::invoke_result<*F*, *Ts*...>::type **operator()**(*F* &&*f*, *Ts*&&... *vs*) **const**

template<typename **R**>
struct invoke_r
#include <invoke.hpp>

Public Functions

template<typename **F**, typename ...**Ts**>
constexpr R **operator()** (*F &&f*, *Ts&&... vs*) **const**

Header `hpx/functional/invoke_fused.hpp`

namespace hpx

namespace util

Functions

template<typename **F**, typename **Tuple**>
constexpr detail::invoke_fused_result<*F*, *Tuple*>::type **invoke_fused** (*F &&f*, *Tuple &&t*)
Invokes the given callable object f with the content of the sequenced type t (tuples, pairs)

Return The result of the callable object when it's called with the content of the given sequenced type.
Note This function is similar to `std::apply` (C++17)
Parameters
- `f`: Must be a callable object. If f is a member function pointer, the first argument in the sequenced type will be treated as the callee (this object).
- `t`: A type which is content accessible through a call to hpx::util::get.

Exceptions
- `std::exception`: like objects thrown by call to object f with the arguments contained in the sequenceable type t.

template<typename **R**, typename **F**, typename **Tuple**>
constexpr *R* **invoke_fused_r** (*F &&f*, *Tuple &&t*)
Invokes the given callable object f with the content of the sequenced type t (tuples, pairs)

Return The result of the callable object when it's called with the content of the given sequenced type.
Note This function is similar to `std::apply` (C++17)
Parameters
- `f`: Must be a callable object. If f is a member function pointer, the first argument in the sequenced type will be treated as the callee (this object).
- `t`: A type which is content accessible through a call to hpx::util::get.

Exceptions
- `std::exception`: like objects thrown by call to object f with the arguments contained in the sequenceable type t.

Template Parameters
- `R`: The result type of the function when it's called with the content of the given sequenced type.

Header hpx/functional/invoke_result.hpp

namespace hpx

namespace util

Functions

`template<typename T>struct hpx::util::HPX_DEPRECATED_V(1, 5, "result_of is deprecat`

Header hpx/functional/mem_fn.hpp

namespace hpx

namespace util

Functions

template<typename **M**, typename **C**>
constexpr detail::mem_fn<*M* C::*> **mem_fn** (*M C*::* *pm*)

template<typename **R**, typename **C**, typename ...**Ps**>
constexpr detail::mem_fn<*R* (C::*) (*Ps*...) > **mem_fn**
R (*C*::* *pm*)*Ps*...

template<typename **R**, typename **C**, typename ...**Ps**>
constexpr detail::mem_fn<*R* (C::*) (*Ps*...) **const**> **mem_fn**
R (*C*::* *pm*)*Ps*... **const**

Header hpx/functional/one_shot.hpp

namespace hpx

namespace serialization

Functions

template<typename **Archive**, typename **F**>
void **serialize** (*Archive* &*ar*, ::*hpx*::*util*::detail::one_shot_wrapper<*F*> &*one_shot_wrapper*, unsigned int **const** *version* = 0)

namespace util

Functions

template<typename **F**>
constexpr detail::one_shot_wrapper<**typename** *std*::decay<*F*>::type> **one_shot** (*F* &&*f*)

Header `hpx/functional/protect.hpp`

namespace hpx

namespace util

Functions

template<typename **T**>
std::enable_if<*traits*::*is_bind_expression*<**typename** *std*::decay<*T*>::type>::value, detail::protected_bind<**typename** *std*

template<typename **T**>
std::enable_if<!*traits*::*is_bind_expression*<**typename** *std*::decay<*T*>::type>::value, *T*>::type **protect** (*T* &&*v*)

Header `hpx/functional/serialization/serializable_function.hpp`

Header `hpx/functional/serialization/serializable_unique_function.hpp`

Header `hpx/functional/tag_invoke.hpp`

namespace hpx

namespace functional

Typedefs

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::functioanl::tag_invoke. This can be used in a SFINAE context.

using tag_invoke_result_t = **typename** *tag_invoke_result*<Tag, Args...>::type
`hpx::functional::tag_invoke_result_t<Tag, Args...>` evaluates to `hpx::functional::tag_invoke_result_t<Tag, Args...>::type`

Variables

template<typename **Tag**, typename ...**Args**>
constexpr bool **is_tag_invocable_v** = *is_tag_invocable*<*Tag*, *Args*...>::value
hpx::functional::is_tag_invocable_v<Tag, Args...> evaluates to *hpx::functional::is_tag_invocable*<Tag, Args...>::value

template<typename **Tag**, typename ...**Args**>
constexpr bool **is_nothrow_tag_invocable_v** = *is_nothrow_tag_invocable*<*Tag*, *Args*...>::value
hpx::functional::is_tag_invocable_v<Tag, Args...> evaluates to *hpx::functional::is_tag_invocable*<Tag, Args...>::value

template<typename **Tag**, typename ...**Args**>
struct is_nothrow_tag_invocable
#include <tag_invoke.hpp> *hpx::functional::is_nothrow_tag_invocable*<Tag, Args...> is std::true_type if an overload of `tag_invoke(tag, args...)` can be found via ADL and is noexcept.

template<typename **Tag**, typename ...**Args**>
struct is_tag_invocable
#include <tag_invoke.hpp> *hpx::functional::is_tag_invocable*<Tag, Args...> is std::true_type if an overload of `tag_invoke(tag, args...)` can be found via ADL.

template<typename **Tag**>
struct tag
#include <tag_invoke.hpp> *hpx::functional::tag*<Tag> defines a base class that implements the necessary tag dispatching functionality for a given type `Tag`

namespace unspecified

Variables

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 CPO). 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`:

```
namespace mylib {
    inline constexpr
        struct foo_fn final : hpx::functional::tag<foo_fn>
        {
        } foo{};
}
```

Defining an object `bar` which customizes `foo`:

```
struct bar
{
    int x = 42;
```

(continues on next page)

(continued from previous page)

```
    friend constexpr int tag_invoke(mylib::foo_fn, bar const& x)
    {
        return b.x;
    }
};
```

Using the customization point:

```
static_assert(42 == mylib::foo(bar{}), "The answer is 42");
```

Header `hpx/functional/traits/get_action_name.hpp`

Header `hpx/functional/traits/get_function_address.hpp`

template<typename **R**, typename **Obj**, typename ...**Ts**>
struct get_function_address<*R* (Obj::*) (*Ts*...) >
#include <get_function_address.hpp>

Public Static Functions

static *std*::size_t **call** (R (*Obj*::* *f*)) Ts...

template<typename **R**, typename **Obj**, typename ...**Ts**>
struct get_function_address<*R* (Obj::*) (*Ts*...) **const**>
#include <get_function_address.hpp>

Public Static Functions

static *std*::size_t **call** (R (*Obj*::* *f*)) Ts...
const

namespace hpx

namespace traits

template<typename **F**, typename **Enable** = void>
struct get_function_address
#include <get_function_address.hpp>

Public Static Functions

`static std::size_t call(F const &f)`

template<typename **R**, typename ...**Ts**>
struct get_function_address<*R* (*) (*Ts*...)>
#include <get_function_address.hpp>

Public Static Functions

`static std::size_t call(R (*f)) Ts...`

template<typename **R**, typename **Obj**, typename ...**Ts**>
struct get_function_address<*R* (Obj::*) (*Ts*...) **const**>
#include <get_function_address.hpp>

Public Static Functions

`static std::size_t call(R (Obj::* f)) Ts... const`

template<typename **R**, typename **Obj**, typename ...**Ts**>
struct get_function_address<*R* (Obj::*) (*Ts*...)>
#include <get_function_address.hpp>

Public Static Functions

`static std::size_t call(R (Obj::* f)) Ts...`

Header `hpx/functional/traits/get_function_annotation.hpp`

namespace hpx

namespace traits

template<typename **F**, typename **Enable** = void>
struct get_function_annotation
#include <get_function_annotation.hpp>

Public Static Functions

`static char const *call(F const&)`

Header `hpx/functional/traits/is_action.hpp`

Header `hpx/functional/traits/is_bind_expression.hpp`

namespace hpx

namespace traits

template<typename **T**>
struct is_bind_expression : **public** *std*::is_bind_expression<*T*>
#include <is_bind_expression.hpp> Subclassed by hpx::traits::is_bind_expression< T const >

Header `hpx/functional/traits/is_invocable.hpp`

namespace hpx

namespace traits

Functions

template<typename T, typename R = void>struct hpx::traits::HPX_DEPRECATED_V(1, 5, '

Header `hpx/functional/traits/is_placeholder.hpp`

namespace hpx

namespace traits

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>`.

Header `hpx/functional/unique_function.hpp`

Defines

HPX_UTIL_REGISTER_UNIQUE_FUNCTION_DECLARATION(*Sig*, *F*, *Name*)

HPX_UTIL_REGISTER_UNIQUE_FUNCTION(*Sig*, *F*, *Name*)

namespace hpx

namespace util

Typedefs

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*> *#include <unique_function.hpp>*

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*::enabl
unique_function (*F* &&*f*)

template<typename **F**, typename **FD** = **typename** *std*::decay<*F*>::type, typename **Enable1** = **typename** *std*::enabl
unique_function &**operator=** (*F* &&*f*)

Private Types

template<>
using base_type = detail::basic_function<R (Ts...) , false, Serializable>

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.

Header `hpx/futures/future.hpp`

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_futur**

template<typename **T**, typename ...**Ts**>
std::enable_if<*std*::is_constructible<*T*, *Ts*&&...>::value || *std*::is_void<*T*>::value, future<*T*>>::type **make_ready_futur**

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*::*util*::*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*::*util*::*steady_duration* **const** &*rel_time*, *T* &&*init*)

template<typename **Allocator**>
future<void> **make_ready_future_alloc** (*Allocator* **const** &*a*)

std::enable_if<*std*::is_void<T>::value, future<void>>::type **make_ready_future** ()

std::enable_if<*std*::is_void<T>::value, future<void>>::type **make_ready_future_at** (*hpx*::*util*::*steady_time_point* **const** &*abs_time*)

std::enable_if<*std*::is_void<T>::value, future<void>>::type **make_ready_future_after** (*hpx*::*util*::*steady_duration* **const** &*rel_time*)

template<typename **R**>
class future : **public** *hpx*::*lcos*::detail::future_base<*future*<*R*>, *R*>
#include <future.hpp>

Public Types

typedef R **result_type**

typedef *base_type*::shared_state_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

typedef detail::future_base<future<R>, R> **base_type**

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

struct invalidate

Public Functions

template<>
invalidate (future *&f*)

template<>
~invalidate ()

Public Members

template<>
future &**f_**

template<typename **R**>
class shared_future : **public** *hpx*::*lcos*::detail::future_base<*shared_future*<*R*>, *R*>
#include <future.hpp>

Public Types

typedef R **result_type**

typedef *base_type*::shared_state_type **shared_state_type**

Public Functions

shared_future()

shared_future(*shared_future* **const** &*other*)

shared_future(*shared_future* &&*other*)

shared_future(future<R> &&*other*)

shared_future(future<*shared_future*> &&*other*)

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)

~shared_future()

shared_future &**operator=**(shared_future **const** &*other*)

shared_future &**operator=**(shared_future &&*other*)

hpx::*traits*::future_traits<shared_future>::result_type **get**() **const**

hpx::*traits*::future_traits<shared_future>::result_type **get**(*error_code* &*ec*) **const**

template<typename **F**>
decltype(auto) **then**(*F* &&*f*, *error_code* &*ec* = *throws*) **const**

template<typename **T0**, typename **F**>
decltype(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

friend hpx::lcos::hpx::traits::future_access

namespace serialization

Functions

template<typename **Archive**, typename **T**>
void **serialize**(*Archive* &*ar*, ::*hpx*::*lcos*::future<*T*> &*f*, unsigned *version*)

template<typename **Archive**, typename **T**>
void **serialize**(*Archive* &*ar*, ::*hpx*::*lcos*::shared_future<*T*> &*f*, unsigned *version*)

Header `hpx/futures/future_fwd.hpp`

Header `hpx/futures/futures_factory.hpp`

namespace hpx

namespace lcos

namespace local

template<typename **Result**, bool **Cancelable**>
class futures_factory<*Result*(), *Cancelable*>
#include <futures_factory.hpp>

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** *hpx*::*util*::decay
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_type* **apply**(**const** char **annotation* = "futures_factory::apply", *launch policy* = *launch*::async, *threads*::*thread_priority priority* = *threads*::*thread_priority_default*, *threads*::*thread_stacksize stacksize* = *threads*::*thread_stacksize_default*, *threads*::*thread_schedule_hint schedulehint* = *threads*::*thread_schedule_hint*(), *error_code* &*ec* = *throws*) **const**

threads::*thread_id_type* **apply**(*threads*::*thread_pool_base* **pool*, **const** char **annotation* = "futures_factory::apply", *launch policy* = *launch*::async, *threads*::*thread_priority priority* = *threads*::*thread_priority_default*, *threads*::*thread_stacksize stacksize* = *threads*::*thread_stacksize_default*, *threads*::*thread_schedule_hint schedulehint* = *threads*::*thread_schedule_hint*(), *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_**

Header `hpx/futures/packaged_continuation.hpp`

Header `hpx/futures/traits/acquire_future.hpp`

namespace hpx

namespace traits

struct acquire_future_disp

#include <acquire_future.hpp>

Public Functions

template<typename **T**>
acquire_future<*T*>::type **operator()**(*T* &&*t*) **const**

Header hpx/futures/traits/acquire_shared_state.hpp

namespace hpx

namespace traits

struct acquire_shared_state_disp
#include <acquire_shared_state.hpp>

Public Functions

template<typename **T**>
acquire_shared_state<*T*>::type **operator()**(*T* &&*t*) **const**

Header hpx/futures/traits/future_access.hpp

template<typename **R**>
struct future_access<*lcos*::future<*R*>>
#include <future_access.hpp>

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**(**typename** detail::shared_state_ptr_for<*lcos*::future<*lcos*::future<R>>>::type **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**(**typename** detail::shared_state_ptr_for<*lcos*::future<*lcos*::future<R>>>::type &&*shared_state*)

template<typename **SharedState**>
static *lcos*::future<R> **create**(*SharedState* **shared_state*, bool *addref* = true)

static *traits*::detail::shared_state_ptr<R>::type **const** &**get_shared_state**(*lcos*::future<R> **const** &*f*)

static *traits*::detail::shared_state_ptr<R>::type::element_type ***detach_shared_state**(*lcos*::future<R> &&*f*)

template<typename **R**>

struct future_access<*lcos*::shared_future<*R*>>
#include <future_access.hpp>

Public Static Functions

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**(**typename** detail::shared_state_ptr_for<*lcos*::shared_future<*lcos*::future<R>>>::typ **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**(**typename** detail::shared_state_ptr_for<*lcos*::shared_future<*lcos*::future<R>>>::typ &&*shared_state*)

template<typename **SharedState**>
static *lcos*::shared_future<R> **create**(*SharedState* **shared_state*, bool *addref* = true)

static *traits*::detail::shared_state_ptr<R>::type **const** &**get_shared_state**(*lcos*::shared_future<R> **const** &*f*)

static *traits*::detail::shared_state_ptr<R>::type::element_type ***detach_shared_state**(*lcos*::shared_future<R> **const** &*f*)

namespace hpx

namespace traits

template<typename **R**>
struct future_access<*lcos*::future<*R*>>
#include <future_access.hpp>

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**(**typename** detail::shared_state_ptr_for<*lcos*::future<*lcos*::future<R>>>::type **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**(**typename** detail::shared_state_ptr_for<*lcos*::future<*lcos*::future<R>>>::type &&*shared_state*)

template<typename **SharedState**>
static *lcos*::future<R> **create**(*SharedState* **shared_state*, bool *addref* = true)

static *traits*::detail::shared_state_ptr<R>::type **const** &**get_shared_state**(*lcos*::future<R> **const** *&f*)

static *traits*::detail::shared_state_ptr<R>::type::element_type ***detach_shared_state**(*lcos*::future<R> *&&f*)

template<typename **R**>
struct future_access<*lcos*::shared_future<*R*>>
#include <future_access.hpp>

Public Static Functions

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**(**typename** detail::shared_state_ptr_for<*lcos*::shared_future<*lcos*::future< **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**(**typename** detail::shared_state_ptr_for<*lcos*::shared_future<*lcos*::future< *&&shared_state*)

template<typename **SharedState**>
static *lcos*::shared_future<R> **create**(*SharedState* **shared_state*, bool *addref* = true)

static *traits*::detail::shared_state_ptr<R>::type **const** &**get_shared_state**(*lcos*::shared_future<R> **const** *&f*)

static *traits*::detail::shared_state_ptr<R>::type::element_type ***detach_shared_state**(*lcos*::shared_future<R> **const** *&f*)

Header `hpx/futures/traits/future_then_result.hpp`

Header `hpx/futures/traits/future_traits.hpp`

template<typename **R**>
struct future_traits<*lcos*::future<*R*>>
#include <future_traits.hpp>

Public Types

typedef R **type**

typedef R **result_type**

template<typename **R**>
struct future_traits<*lcos*::shared_future<*R*>>
#include <future_traits.hpp>

Public Types

typedef R **type**

typedef R **const** &**result_type**

template<>
struct future_traits<*lcos*::shared_future<void>>
#include <future_traits.hpp>

Public Types

typedef void **type**

typedef void **result_type**

namespace hpx

namespace traits

template<typename **R**>
struct future_traits<*lcos*::future<*R*>>
#include <future_traits.hpp>

Public Types

typedef R **type**

typedef R **result_type**

template<typename **R**>
struct future_traits<*lcos*::shared_future<*R*>>
#include <future_traits.hpp>

Public Types

typedef R **type**

typedef R **const** &**result_type**

template<>
struct future_traits<*lcos*::shared_future<void>>
#include <future_traits.hpp>

Public Types

typedef void **type**

typedef void **result_type**

Header `hpx/futures/traits/get_remote_result.hpp`

namespace hpx

namespace traits

template<typename **Result**, typename **RemoteResult**, typename **Enable** = void>
struct get_remote_result
#include <get_remote_result.hpp>

Public Static Functions

static Result **call**(RemoteResult **const** &*rhs*)

static Result **call**(RemoteResult &&*rhs*)

template<typename **Result**>
struct get_remote_result<*Result*, *Result*>
#include <get_remote_result.hpp>

Public Static Functions

static Result **const** &**call**(Result **const** &*rhs*)

static Result &&**call**(Result &&*rhs*)

Header `hpx/futures/traits/is_future.hpp`

namespace hpx

namespace traits

template<typename **Future**>
struct is_future : **public** *hpx*::*traits*::detail::is_future_customization_point<*Future*>
#include <is_future.hpp> Subclassed by hpx::traits::is_ref_wrapped_future< std::reference_wrapper< Future > >

Header `hpx/futures/traits/is_future_range.hpp`

namespace hpx

namespace traits

template<typename **R**>
struct future_range_traits<*R*, true>
#include <is_future_range.hpp>

Public Types

typedef range_traits<R>::value_type **future_type**

Header `hpx/futures/traits/is_future_tuple.hpp`

Header `hpx/futures/traits/promise_local_result.hpp`

template<>
struct promise_local_result<*util*::*unused_type*>
#include <promise_local_result.hpp>

Public Types

typedef void **type**

namespace hpx

namespace traits

template<typename **Result**, typename **Enable** = void>
struct promise_local_result
#include <promise_local_result.hpp>

Public Types

typedef Result **type**

template<>
struct promise_local_result<*util*::*unused_type*>
#include <promise_local_result.hpp>

Public Types

typedef void **type**

Header `hpx/futures/traits/promise_remote_result.hpp`

namespace hpx

namespace traits

template<typename **Result**, typename **Enable** = void>
struct promise_remote_result
#include <promise_remote_result.hpp>

Public Types

typedef Result **type**

template<>
struct promise_remote_result<void>
#include <promise_remote_result.hpp>

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.

Header `hpx/hardware/bit_manipulation.hpp`

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
#include <bit_manipulation.hpp>

Public Static Functions

static T **shl** (T *x*)

static T **shr** (T *x*)

template<typename **T**>
struct unbounded_shifter<0, *T*>
#include <bit_manipulation.hpp>

Public Static Functions

static T **shl** (T *x*)

static T **shr** (T *x*)

Header hpx/hardware/cpuid.hpp

Header hpx/hardware/cpuid/linux_x86.hpp

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*)

struct cpuid_register

#include <linux_x86.hpp>

Public Types

enum info

Values:

eax = 0

ebx = 1

ecx = 2

edx = 3

Header hpx/hardware/cpuid/msvc.hpp

Header hpx/hardware/timestamp.hpp

Header hpx/hardware/timestamp/bgq.hpp

Header hpx/hardware/timestamp/linux_generic.hpp

Header hpx/hardware/timestamp/linux_x86_32.hpp

namespace hpx

namespace util

namespace hardware

Functions

std::uint64_t **timestamp**()

Header hpx/hardware/timestamp/linux_x86_64.hpp

Header hpx/hardware/timestamp/msvc.hpp

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.

Header hpx/hashing/fibhash.hpp

namespace hpx

namespace util

Functions

template<*std*::uint64_t **N**>
constexpr *std*::uint64_t **fibhash**(*std*::uint64_t *i*)

Header hpx/hashing/jenkins_hash.hpp

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**

this 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 **k*, *std*::size_t *length*) **const**

Private Members

size_type **seed_**

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.

Header `hpx/algorithm.hpp`

Header `hpx/any.hpp`

Header `hpx/barrier.hpp`

Header `hpx/channel.hpp`

Header `hpx/chrono.hpp`

Header hpx/distributed/future.hpp

Header hpx/distributed/runtime.hpp

Header hpx/execution.hpp

Header hpx/functional.hpp

Header hpx/future.hpp

Header hpx/hpx.hpp

Header hpx/include/actions.hpp

Header hpx/include/agas.hpp

Header hpx/include/applier.hpp

Header hpx/include/apply.hpp

Header hpx/include/async.hpp

Header hpx/include/bind.hpp

Header hpx/include/client.hpp

Header hpx/include/components.hpp

Header hpx/include/compression.hpp

Header hpx/include/compression_registration.hpp

Header hpx/include/compute.hpp

Header hpx/include/dataflow.hpp

Header hpx/include/datapar.hpp

Header hpx/include/future.hpp

Header hpx/include/lcos.hpp

Header hpx/include/lcos_local.hpp

Header hpx/include/naming.hpp

Header hpx/include/parallel_adjacent_difference.hpp

Header hpx/include/parallel_adjacent_find.hpp

Header hpx/include/parallel_algorithm.hpp

Header hpx/include/parallel_all_any_none_of.hpp

Header hpx/include/parallel_container_algorithm.hpp

Header hpx/include/parallel_copy.hpp

Header hpx/include/parallel_count.hpp

Header hpx/include/parallel_destroy.hpp

Header hpx/include/parallel_equal.hpp

Header hpx/include/parallel_exception_list.hpp

Header hpx/include/parallel_execution.hpp

Header hpx/include/parallel_execution_policy.hpp

Header hpx/include/parallel_executor_information.hpp

Header hpx/include/parallel_executor_parameters.hpp

Header hpx/include/parallel_executors.hpp

Header hpx/include/parallel_fill.hpp

Header hpx/include/parallel_find.hpp

Header hpx/include/parallel_for_each.hpp

Header hpx/include/parallel_for_loop.hpp

Header hpx/include/parallel_generate.hpp

Header hpx/include/parallel_is_heap.hpp

Header hpx/include/parallel_is_partitioned.hpp

Header hpx/include/parallel_is_sorted.hpp

Header hpx/include/parallel_lexicographical_compare.hpp

Header hpx/include/parallel_memory.hpp

Header hpx/include/parallel_merge.hpp

Header hpx/include/parallel_minmax.hpp

Header hpx/include/parallel_mismatch.hpp

Header hpx/include/parallel_move.hpp

Header hpx/include/parallel_numeric.hpp

Header hpx/include/parallel_partition.hpp

Header hpx/include/parallel_reduce.hpp

Header hpx/include/parallel_remove.hpp

Header hpx/include/parallel_remove_copy.hpp

Header hpx/include/parallel_replace.hpp

Header hpx/include/parallel_reverse.hpp

Header hpx/include/parallel_rotate.hpp

Header hpx/include/parallel_scan.hpp

Header hpx/include/parallel_search.hpp

Header hpx/include/parallel_set_operations.hpp

Header hpx/include/parallel_sort.hpp

Header hpx/include/parallel_swap_ranges.hpp

Header hpx/include/parallel_task_block.hpp

Header hpx/include/parallel_transform.hpp

Header hpx/include/parallel_transform_reduce.hpp

Header hpx/include/parallel_transform_scan.hpp

Header hpx/include/parallel_uninitialized_copy.hpp

Header hpx/include/parallel_uninitialized_default_construct.hpp

Header hpx/include/parallel_uninitialized_fill.hpp

Header hpx/include/parallel_uninitialized_move.hpp

Header hpx/include/parallel_uninitialized_value_construct.hpp

Header hpx/include/parallel_unique.hpp

Header hpx/include/parcel_coalescing.hpp

Header hpx/include/parcelset.hpp

Header hpx/include/performance_counters.hpp

Header hpx/include/plain_actions.hpp

Header hpx/include/resource_partitioner.hpp

Header hpx/include/run_as.hpp

Header hpx/include/runtime.hpp

Header hpx/include/serialization.hpp

Header hpx/include/sync.hpp

Header hpx/include/thread_executors.hpp

Header hpx/include/threadmanager.hpp

Header hpx/include/threads.hpp

Header hpx/include/traits.hpp

Header hpx/include/util.hpp

Header hpx/latch.hpp

Header hpx/local/execution.hpp

Header hpx/local/future.hpp

Header `hpx/local/runtime.hpp`

Header `hpx/memory.hpp`

Header `hpx/numeric.hpp`

Header `hpx/optional.hpp`

Header `hpx/runtime.hpp`

Header `hpx/task_block.hpp`

namespace hpx

Typedefs

using task_cancelled_exception = *hpx*::*parallel*::task_canceled_exception

Header `hpx/tuple.hpp`

Header `hpx/type_traits.hpp`

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.

Header `hpx/hpx_finalize.hpp`

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

- `shutdown_timeout`: This parameter allows to specify a timeout (in microseconds), specifying how long any of the connected localities should wait for pending tasks to be executed. After this timeout, all suspended HPX-threads will be aborted. Note, that this function will not abort any running HPX-threads. In any case the shutdown will not proceed as long as there is at least one pending/running HPX-thread.

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 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.

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.

`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*.

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.

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.

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`).

Header `hpx/hpx_init.hpp`

namespace hpx

Functions

int **init**(*util*::*function_nonser*<int) *hpx*::*program_options*::*variables_map*&
> **const** &*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**(*util*::*function_nonser*<int) int, char**
> **const** &*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** (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)

int **init** (*std*::nullptr_t *f*, int *argc*, char ***argv*, *init_params* **const** &*params* = *init_params*())

Main entry point for launching the HPX runtime system.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. This overload will not call `hpx_main`.

This is the main entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread.

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** (*init_params* **const** *&params* = *init_params*())

Main entry point for launching the HPX runtime system.

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

This is a simplified main entry point, which can be used to set up the runtime for an HPX application (the runtime system will be set up in console mode or worker mode depending on the command line settings).

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)

Header `hpx/hpx_init_impl.hpp`

Header `hpx/hpx_init_params.hpp`

namespace hpx

struct init_params

#include <hpx_init_params.hpp> Parameters used to initialize the HPX runtime through hpx::init and hpx::start.

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::*runtime_mode* **mode** = ::*hpx*::*runtime_mode*::default_

hpx::*resource*::*partitioner_mode* **rp_mode** = ::*hpx*::*resource*::*mode_default*

hpx::*resource*::rp_callback_type **rp_callback**

Header `hpx/hpx_main_winsocket.hpp`

Header `hpx/hpx_start.hpp`

namespace hpx

Functions

bool **start** (*util*::*function_nonser*<int) *hpx*::*program_options*::*variables_map*& > **const** &*f*, int *argc*, char ***argv*, *init_params* **const** &*params* = *init_params*()Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution.

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 parameter`mode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the hpx::start function (See documentation of hpx::init_params)

bool **start** (*util*::*function_nonser*<int) int, char**
> **const** &*f*, int *argc*, char ****argv*, *init_params* **const** &*params* = *init_params*()Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution.

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 parameter`mode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the hpx::start function (See documentation of hpx::init_params)

bool **start** (int *argc*, char ****argv*, *init_params* **const** &*params* = *init_params*())
Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution.

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 parameter`mode`.

Parameters

- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).

- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the hpx::start function (See documentation of hpx::init_params)

bool **start** (*std*::nullptr_t *f*, int *argc*, char ***argv*, *init_params* **const** &*params* = *init_params*())

Main non-blocking entry point for launching the HPX runtime system.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as a HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution. This overload will not call `hpx_main`.

This is the main, non-blocking entry point for any HPX application. This function (or one of its overloads below) should be called from the users `main()` function. It will set up the HPX runtime environment and schedule the function given by `f` as an HPX thread. It will return immediately after that. Use `hpx::wait` and *`hpx::stop`* to synchronize with the runtime system's execution.

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 parameter`mode`.

Parameters

- `f`: [in] The function to be scheduled as an HPX thread. Usually this function represents the main entry point of any HPX application. If `f` is `nullptr` the HPX runtime environment will be started without invoking `f`.
- `argc`: [in] The number of command line arguments passed in `argv`. This is usually the unchanged value as passed by the operating system (to `main()`).
- `argv`: [in] The command line arguments for this application, usually that is the value as passed by the operating system (to `main()`).
- `params`: [in] The parameters to the hpx::start function (See documentation of hpx::init_params)

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)

Header hpx/hpx_start_impl.hpp

Header hpx/hpx_suspend.hpp

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.

Header `hpx/hpx_user_main_config.hpp`

namespace hpx_startup

Functions

std::vector<*std*::string> **user_main_config** (*std*::vector<*std*::string> **const** &*cfg*)

Variables

std::vector<*std*::string> (***user_main_config_function**) (*std*::vector<*std*::string> **const**&)

Header `hpx/init.hpp`

io_service

The contents of this module can be included with the header `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 `hpx/modules/io_service.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.

Header `hpx/io_service/io_service_pool.hpp`

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.

void **wait**()
Wait for all work to be done.

bool **stopped**()

boost::asio::io_service &**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.

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<*boost*::asio::io_service>

using work_type = *boost*::asio::io_service::work

Private Functions

work_type **initialize_work**(*boost*::asio::io_service &*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.

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.

barrier **wait_barrier_**

barrier **continue_barrier_**

Header `hpx/io_service/io_service_thread_pool.hpp`

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.

Header hpx/iterator_support/counting_iterator.hpp

template<typename **Incrementable**, typename **CategoryOrTraversal**, typename **Difference**>
class counting_iterator<*Incrementable*, *CategoryOrTraversal*, *Difference*, **typename** *std*::enable_if<*std*::is_integral<*Incre*

#include <counting_iterator.hpp>

Public Functions

counting_iterator()

counting_iterator(*counting_iterator* **const** &*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 **Enab**
class counting_iterator : **public** *hpx*::*util*::iterator_adaptor<*counting_iterator*<*Incrementable*, *CategoryOrTra*
#include <counting_iterator.hpp>

Public Functions

counting_iterator()

counting_iterator(*counting_iterator* **const** &*rhs*)

counting_iterator(Incrementable *x*)

Private Types

template<>
using base_type = **typename** detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>

Private Functions

base_type::reference **dereference**() **const**

Friends

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_inte
#include <counting_iterator.hpp>

Public Functions

counting_iterator()

counting_iterator(*counting_iterator* **const** &*rhs*)

counting_iterator(Incrementable *x*)

Private Types

template<>
using base_type = **typename** detail::counting_iterator_base<Incrementable, CategoryOrTraversal, Difference>

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 hpx::util::iterator_core_access

Header `hpx/iterator_support/iterator_adaptor.hpp`

namespace hpx

namespace util

template<typename **Derived**, typename **Base**, typename **Value** = void, typename **Category** = void, typename **Refer**
class iterator_adaptor : **public** *hpx*::*util*::iterator_facade<*Derived*, value_type, iterator_category, reference_typ

#include <iterator_adaptor.hpp> Subclassed by *hpx::util::counting_iterator< Incrementable, CategoryOrTraversal, Difference, Enable >*, hpx::util::counting_iterator< Incrementable, CategoryOrTraversal, Difference, typename std::enable_if< std::is_integral< Incrementable >::value >::type >, *hpx::util::transform_iterator< Iterator, Transformer, Reference, Value, Category, Difference >*

Public Types

typedef Base **base_type**

Public Functions

iterator_adaptor()

iterator_adaptor(Base **const** &*iter*)

Base **const** &**base**() **const**

Protected Types

typedef *hpx*::*util*::detail::iterator_adaptor_base<Derived, Base, Value, Category, Reference, Difference, Pointer>::ty

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
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<*traits*::is_bidirectional_iterat
void **decrement**()

template<typename **OtherDerived**, typename **OtherIterator**, typename **V**, typename **C**, typename **R**, typename
base_adaptor_type::difference_type **distance_to**(iterator_adaptor<*OtherDerived*, *OtherIterator*, *V*, *C*, *R*, *D*, *P*> **const** &*y*) **const**

Private Members

Base **iterator_**

Friends

friend hpx::util::hpx::util::iterator_core_access

Header `hpx/iterator_support/iterator_facade.hpp`

Defines

HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD (*prefix*, *op*, *result_type*)

namespace hpx

namespace util

Functions

template<typename **Derived**, typename **T**, typename **Category**, typename **Reference**, typename **Distance**, typena
util::detail::postfix_increment_result<*Derived*, **typename** *Derived*::value_type, **typename** *Derived*::reference>::type **op**

hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline, bool)

hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline, !, bool)

hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline)

hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline, <=, bool)

hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline, >=, bool)

hpx::util::HPX_UTIL_ITERATOR_FACADE_INTEROP_HEAD(inline, -, typename std::iterator_

template<typename **Derived**, typename **T**, typename **Category**, typename **Reference**, typename **Distance**, typena
Derived **operator+** (iterator_facade<*Derived*, *T*, *Category*, *Reference*, *Distance*, *Pointer*> **const** &*it*, **typename** *Derived*::difference_type *n*)

template<typename **Derived**, typename **T**, typename **Category**, typename **Reference**, typename **Distance**, typena
Derived **operator+** (**typename** *Derived*::difference_type *n*, iterator_facade<*Derived*, *T*, *Category*, *Reference*, *Distance*, *Pointer*> **const** &*it*)

class iterator_core_access

#include <iterator_facade.hpp>

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** =
struct iterator_facade : **public** *hpx*::*util*::detail::iterator_facade_base<*Derived*, *T*, *Category*, *Reference*, *Distan*
#include <iterator_facade.hpp> Subclassed by *hpx::util::iterator_adaptor< Derived, Base, Value, Category, Reference, Difference, Pointer >*

Public Functions

iterator_facade ()

Protected Types

typedef iterator_facade<Derived, T, Category, Reference, Distance, Pointer> **iterator_adaptor_**

Private Types

typedef detail::iterator_facade_base<Derived, T, Category, Reference, Distance, Pointer> **base_type**

Header `hpx/iterator_support/iterator_range.hpp`

namespace hpx

namespace util

Functions

template<typename **Range**, typename **Iterator** = **typename** *traits*::range_iterator<*Range*>::type, typename **Sentine**
std::enable_if<*traits*::is_range<*Range*>::value, iterator_range<*Iterator*, *Sentinel*>>::type **make_iterator_range**(*Rang* &*r*)

template<typename **Range**, typename **Iterator** = **typename** *traits*::range_iterator<*Range* **const**>::type, typename **S**
std::enable_if<*traits*::is_range<*Range*>::value, iterator_range<*Iterator*, *Sentinel*>>::type **make_iterator_range**(*Rang* **con** &*r*)

template<typename **Iterator**, typename **Sentinel** = *Iterator*>
std::enable_if<*traits*::is_iterator<*Iterator*>::value, iterator_range<*Iterator*, *Sentinel*>>::type **make_iterator_range**(*It* *it* *er* *a* *to* *Se* *ti* *se* *ti*

template<typename **Iterator**, typename **Sentinel** = *Iterator*>
class iterator_range

#include <iterator_range.hpp>

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**

Header `hpx/iterator_support/range.hpp`

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** =
constexpr *std*::size_t **size**(*C* **const** &*c*)

template<typename **C**, typename **Iterator** = **typename** detail::iterator<*C* **const**>::type, typename **Sentinel** =
constexpr bool **empty**(*C* **const** &*c*)

Header `hpx/iterator_support/traits/is_iterator.hpp`

Header `hpx/iterator_support/traits/is_range.hpp`

namespace hpx

namespace traits

template<typename **R**>
struct range_traits<*R*, true> : **public** *std*::iterator_traits<*util*::detail::iterator<*R*>::type>
#include <is_range.hpp>

Public Types

typedef *util*::detail::iterator<R>::type **iterator_type**

typedef *util*::detail::sentinel<R>::type **sentinel_type**

Header `hpx/iterator_support/traits/is_sentinel_for.hpp`

namespace hpx

namespace traits

Variables

template<typename Sent, typename Iter>HPX_INLINE_CONSTEXPR_VARIABLE bool hpx::trait

Header `hpx/iterator_support/transform_iterator.hpp`

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 **Categ**
class transform_iterator : **public** *hpx*::*util*::iterator_adaptor<*transform_iterator*<*Iterator*, *Transformer*, *Refere*
#include <transform_iterator.hpp>

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

transform_iterator (*transform_iterator*<*OtherIterator*, *OtherTransformer*, *OtherReference*, *OtherValue*, *OtherCategory*, *OtherDifference*> **const** &*t*, **typename** *std*::enable_if<*std*::is_convertible<*OtherIterator*, Iterator>::value && *std*::is_convertible<*OtherTransformer*, Transformer>::value && *std*::is_convertible<*OtherCategory*, Category>::value && *std*::is_convertible<*OtherDifference*, Difference>::value>::type* = nullptr)

Transformer **const** &**transformer**() **const**

Private Types

typedef detail::transform_iterator_base<Iterator, Transformer, Reference, Value, Category, Difference>::type **base**

Private Functions

base_type::reference **dereference**() **const**

Private Members

Transformer **transformer_**

Friends

friend hpx::util::hpx::util::iterator_core_access

Header hpx/iterator_support/zip_iterator.hpp

template<typename **F**, typename ...**Ts**>
struct lift_zipped_iterators<*F*, *util*::zip_iterator<*Ts*...>>
#include <zip_iterator.hpp>

Public Types

typedef *util*::zip_iterator<Ts...>::iterator_tuple_type **tuple_type**

typedef *util*::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*...>, *util*::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*...>>
#include <zip_iterator.hpp>

Public Types

typedef *util*::zip_iterator<Ts...>::iterator_tuple_type **tuple_type**

typedef *util*::tuple<**typename** element_result_of<**typename** F::template apply<Ts>, Ts>::type...> **result_**

Public Static Functions

template<*std*::size_t... **Is**, typename ...**Ts_**>
static *result_type* **call**(*util*::*index_pack*<*Is*...>, *util*::tuple<*Ts_*...> **const** &*t*)

template<typename ...**Ts_**>
static *result_type* **call**(*util*::zip_iterator<*Ts_*...> **const** &*iter*)

namespace util

Functions

template<typename ...**Ts**>
zip_iterator<**typename** decay<*Ts*>::type...> **make_zip_iterator**(*Ts*&&... *vs*)

template<typename ...**Ts**>
class zip_iterator : **public** *hpx*::*util*::detail::zip_iterator_base<tuple<*Ts*...>, *zip_iterator*<*Ts*...>>
#include <zip_iterator.hpp>

Public Functions

zip_iterator()

zip_iterator(Ts **const**&... *vs*)

zip_iterator(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_*...>::iter

template<typename ...**Ts_**>
std::enable_if<*std*::is_assignable<**typename** zip_iterator::iterator_tuple_type&, **typename** zip_iterator<*Ts_*...>::iter

Private Types

typedef detail::zip_iterator_base<tuple<Ts...>, zip_iterator<Ts...>> **base_type**

template<typename ...**Ts**>
class zip_iterator<tuple<*Ts*...>> : **public** *hpx*::*util*::detail::zip_iterator_base<tuple<*Ts*...>, zip_iterator<tuple<*Ts*
#include <zip_iterator.hpp>

Public Functions

zip_iterator()

zip_iterator(Ts **const**&... *vs*)

zip_iterator(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_*...>::iter

template<typename ...**Ts_**>
std::enable_if<*std*::is_assignable<**typename** zip_iterator::iterator_tuple_type&, **typename** zip_iterator<*Ts_*...>::iter

Private Types

typedef detail::zip_iterator_base<tuple<Ts...>, zip_iterator<tuple<Ts...>>> **base_type**

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.

Header hpx/itt_notify/thread_name.hpp

Header hpx/modules/itt_notify.hpp

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_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

typedef void ***__itt_heap_function**

Functions

void **itt_sync_create** (void*, **const** char*, **const** char*)

void **itt_sync_rename** (void*, **const** char*)

void **itt_sync_prepare** (void*)

void **itt_sync_acquired** (void*)

void **itt_sync_cancel** (void*)

void **itt_sync_releasing** (void*)

void **itt_sync_released** (void*)

void **itt_sync_destroy** (void*)

___itt_caller ***itt_stack_create** ()

void **itt_stack_enter** (___itt_caller*)

void **itt_stack_leave** (___itt_caller*)

void **itt_stack_destroy** (___itt_caller*)

void **itt_frame_begin** (___itt_domain **const***, ___itt_id*)

void **itt_frame_end** (___itt_domain **const***, ___itt_id*)

int **itt_mark_create** (char **const***)

void **itt_mark_off** (int)

void **itt_mark** (int, char **const***)

void **itt_thread_set_name** (char **const***)

void **itt_thread_ignore** ()

void **itt_task_begin** (___itt_domain **const***, ___itt_string_handle*)

void **itt_task_begin** (___itt_domain **const***, ___itt_id*, ___itt_string_handle*)

void **itt_task_end** (___itt_domain **const***) ___itt_domain ***itt_domain_create**(char **const***)

___itt_string_handle ***itt_string_handle_create**(char **const***)

___itt_id ***itt_make_id**(void*, unsigned long)

void **itt_id_create**(___itt_domain **const***, ___itt_id*)

void **itt_id_destroy**(___itt_id*)

__itt_heap_function **itt_heap_function_create**(**const** char*, **const** char*)

void **itt_heap_allocate_begin**(*__itt_heap_function*, *std*::size_t, int)

void **itt_heap_allocate_end**(*__itt_heap_function*, void**, *std*::size_t, int)

void **itt_heap_free_begin**(*__itt_heap_function*, void*)

void **itt_heap_free_end**(*__itt_heap_function*, void*)

void **itt_heap_reallocate_begin**(*__itt_heap_function*, void*, *std*::size_t, int)

void **itt_heap_reallocate_end**(*__itt_heap_function*, void*, void**, *std*::size_t, int)

void **itt_heap_internal_access_begin**()

void **itt_heap_internal_access_end**()

___itt_counter ***itt_counter_create**(char **const***, char **const***)

___itt_counter ***itt_counter_create_typed**(char **const***, char **const***, int)

void **itt_counter_destroy**(___itt_counter*)

void **itt_counter_set_value**(___itt_counter*, void*)

int **itt_event_create**(char **const***, int)

int **itt_event_start**(int)

int **itt_event_end**(int)

void **itt_metadata_add**(___itt_domain*, ___itt_id*, ___itt_string_handle*, *std*::uint64_t **const**&)

void **itt_metadata_add**(___itt_domain*, ___itt_id*, ___itt_string_handle*, double **const**&)

void **itt_metadata_add**(___itt_domain*, ___itt_id*, ___itt_string_handle*, char **const***)

void **itt_metadata_add**(___itt_domain*, ___itt_id*, ___itt_string_handle*, void **const***)

namespace hpx

namespace util

namespace itt

Functions

void **event_tick** (*event* **const**&)

struct caller_context
#include <itt_notify.hpp>

Public Functions

caller_context (*stack_context*&)

~caller_context ()

struct counter
#include <itt_notify.hpp>

Public Functions

counter (char **const***, char **const***)

~counter ()

struct domain
#include <itt_notify.hpp> Subclassed by *hpx::util::itt::thread_domain*

Public Functions

HPX_NON_COPYABLE (*domain*)

domain (char **const***)

domain ()

struct event
#include <itt_notify.hpp>

Public Functions

event (char **const***)

struct frame_context
#include <itt_notify.hpp>

Public Functions

frame_context(*domain* **const**&, *id** = nullptr)

~frame_context()

struct heap_allocate
#include <itt_notify.hpp>

Public Functions

template<typename **T**>
heap_allocate(*heap_function*&, *T***, *std*::size_t, int)

~heap_allocate()

struct heap_free
#include <itt_notify.hpp>

Public Functions

heap_free(*heap_function*&, void*)

~heap_free()

struct heap_function
#include <itt_notify.hpp>

Public Functions

heap_function(char **const***, char **const***)

~heap_function()

struct heap_internal_access
#include <itt_notify.hpp>

Public Functions

heap_internal_access()

~heap_internal_access()

struct id
#include <itt_notify.hpp>

Public Functions

id(*domain* **const**&, void*, unsigned long = 0)

~id()

struct mark_context
#include <itt_notify.hpp>

Public Functions

mark_context(char **const***)

~mark_context()

struct mark_event
#include <itt_notify.hpp>

Public Functions

mark_event(*event* **const**&)

~mark_event()

struct stack_context
#include <itt_notify.hpp>

Public Functions

stack_context()

~stack_context()

struct string_handle
#include <itt_notify.hpp>

Public Functions

string_handle(char **const*** = nullptr)

struct task
#include <itt_notify.hpp>

Public Functions

task (*domain* **const**&, *string_handle* **const**&, *std*::uint64_t)

task (*domain* **const**&, *string_handle* **const**&)

~task ()

struct thread_domain : **public** *hpx*::*util*::*itt*::*domain*
#include <itt_notify.hpp>

Public Functions

HPX_NON_COPYABLE (*thread_domain*)

thread_domain ()

struct undo_frame_context
#include <itt_notify.hpp>

Public Functions

undo_frame_context (*frame_context* **const**&)

~undo_frame_context ()

struct undo_mark_context
#include <itt_notify.hpp>

Public Functions

undo_mark_context (*mark_context* **const**&)

~undo_mark_context ()

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.

Header hpx/distributed/channel.hpp

Header hpx/lcos_distributed/channel.hpp

namespace hpx

namespace lcos

template<typename **T**>
class channel : **public** *components*::client_base<*channel*<*T*>, *lcos*::*server*::channel<*T*>>
#include <channel.hpp>

Public Types

typedef T **value_type**

Public Functions

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<typename **U**, typename **U2** = T>
std::enable_if<!*std*::is_void<*U2*>::value, bool>::type **set**(*launch*::apply_policy, *U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U**, typename **U2** = T>
std::enable_if<!*std*::is_void<*U2*>::value, *hpx*::future<void>>::type **set**(*launch*::async_policy, *U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U**, typename **U2** = T>
std::enable_if<!*std*::is_void<*U2*>::value>::type **set**(*launch*::sync_policy, *U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U**, typename **U2** = T>

std::enable_if<!*std*::is_void<*U2*>::value && !*traits*::is_launch_policy<*U*>::value>::type **set** (*U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value, bool>::type **set** (*launch*::apply_policy, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value, *hpx*::future<void>>::type **set** (*launch*::async_policy, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value>::type **set** (*launch*::sync_policy, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value>::type **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)

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

typedef *components*::client_base<channel<T>, *lcos*::*server*::channel<T>> **base_type**

Private Static Attributes

constexpr *std*::size_t **default_generation** = *std*::size_t(-1)

template<typename **T**, typename **Channel**>
class channel_iterator : **public** *hpx*::*util*::iterator_facade<*channel_iterator*<*T*, *Channel*>, *T* **const**, *std*::input_
#include <channel.hpp>

Public Functions

channel_iterator()

channel_iterator(Channel **const** &*c*)

Private Types

typedef *hpx*::*util*::iterator_facade<channel_iterator<T, Channel>, T **const**, *std*::input_iterator_tag> **base_type**

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

Channel **const** ***channel_**

std::pair<T, bool> **data_**

Friends

friend hpx::lcos::hpx::util::iterator_core_access

template<typename **Channel**>
class channel_iterator<void, *Channel*> : **public** *hpx*::*util*::iterator_facade<channel_iterator<void, *Channel*>, *u*

#include <channel.hpp>

Public Functions

channel_iterator()

channel_iterator(Channel **const** &*c*)

Private Types

typedef *hpx*::*util*::iterator_facade<channel_iterator<void, Channel>, *util*::*unused_type* **const**, *std*::input_iterator_t

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** *components*::client_base<*receive_channel*<*T*>, *lcos*::*server*::channel<*T*>>
#include <channel.hpp>

Public Types

typedef T **value_type**

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**

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**

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

typedef *components*::client_base<receive_channel<T>, *lcos*::*server*::channel<T>> **base_type**

Private Static Attributes

constexpr *std*::size_t **default_generation** = *std*::size_t(-1)

template<typename **T**>
class send_channel : **public** *components*::client_base<*send_channel*<*T*>, *lcos*::*server*::channel<*T*>>
#include <channel.hpp>

Public Types

typedef T **value_type**

Public Functions

send_channel()

send_channel(channel<T> **const** &*c*)

send_channel(*hpx*::future<*naming*::id_type> &&*id*)

send_channel(*hpx*::shared_future<*naming*::id_type> &&*id*)

send_channel(*hpx*::shared_future<*naming*::id_type> **const** &*id*)

template<typename **U**, typename **U2** = T>
std::enable_if<!*std*::is_void<*U2*>::value, bool>::type **set**(*launch*::apply_policy, *U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U**, typename **U2** = T>
std::enable_if<!*std*::is_void<*U2*>::value, *hpx*::future<void>>::type **set**(*launch*::async_policy, *U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U**, typename **U2** = T>
std::enable_if<!*std*::is_void<*U2*>::value>::type **set**(*launch*::sync_policy, *U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U**, typename **U2** = T>
std::enable_if<!*std*::is_void<*U2*>::value && !*traits*::is_launch_policy<*U*>::value>::type **set**(*U val*, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value, bool>::type **set** (*launch*::apply_policy, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value, *hpx*::future<void>>::type **set** (*launch*::async_policy, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value>::type **set** (*launch*::sync_policy, *std*::size_t *generation* = *default_generation*)

template<typename **U** = T>
std::enable_if<*std*::is_void<*U*>::value>::type **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

typedef *components*::client_base<send_channel<T>, *lcos*::*server*::channel<T>> **base_type**

Private Static Attributes

constexpr *std*::size_t **default_generation** = *std*::size_t(-1)

Header `hpx/lcos_distributed/server/channel.hpp`

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 **RemoteType**>
class channel : **public** *lcos*::base_lco_with_value<*T*, *RemoteType*, *traits*::detail::component_tag>, **public** *co*
#include <channel.hpp>

Public Types

typedef *lcos*::base_lco_with_value<T, RemoteType, *traits*::detail::component_tag> **base_type_holder**

typedef *base_type*::wrapping_type **wrapping_type**

Public Functions

channel()

void **set_value**(RemoteType &&*result*)

void **set_exception**(*std*::exception_ptr **const**&)

result_type **get_value**()

result_type **get_value**(*error_code* &*ec*)

hpx::future<T> **get_generation**(*std*::size_t *generation*)

HPX_DEFINE_COMPONENT_DIRECT_ACTION(channel, *get_generation*)

void **set_generation**(RemoteType &&*value*, *std*::size_t *generation*)

HPX_DEFINE_COMPONENT_DIRECT_ACTION(channel, *set_generation*)

std::size_t **close**(bool *force_delete_entries*)

HPX_DEFINE_COMPONENT_ACTION(channel, *close*)

Public Static Functions

static *components*::component_type **get_component_type**()

static void **set_component_type**(*components*::component_type *type*)

Private Types

typedef *components*::component_base<channel> **base_type**

typedef *std*::conditional<*std*::is_void<T>::value, *util*::*unused_type*, T>::type **result_type**

Private Members

lcos::*local*::channel<*result_type*> **channel_**

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.

Header `hpx/lcos_local/and_gate.hpp`

namespace hpx

namespace lcos

namespace local

struct and_gate : **public** *hpx*::*lcos*::*local*::base_and_gate<*no_mutex*>
#include <and_gate.hpp>

Public Functions

and_gate (*std*::size_t *count* = 0)

and_gate (*and_gate* &&*rhs*)

and_gate &**operator=** (*and_gate* &&*rhs*)

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*)

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*)

template<typename **Lock**>
bool **set** (*std*::size_t *which*, *Lock* &*l*, *error_code* &*ec* = *throws*)

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
#include <and_gate.hpp>

Public Functions

base_and_gate(*std*::size_t *count* = 0)
This constructor initializes the *base_and_gate* object from the 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
#include <and_gate.hpp>

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_**

Header hpx/lcos_local/channel.hpp

namespace hpx

namespace lcos

namespace local

template<typename **T**>
class channel : **protected** *hpx*::*lcos*::*local*::detail::channel_base<*T*>
#include <channel.hpp>

Public Types

typedef T **value_type**

Public Functions

channel()

Private Types

typedef detail::channel_base<T> **base_type**

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>
#include <channel.hpp>

Public Types

typedef void **value_type**

Public Functions

channel ()

Private Types

typedef detail::channel_base<void> **base_type**

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*::futur

#include <channel.hpp>

Public Functions

channel_async_iterator ()

channel_async_iterator (detail::channel_base<T> **const** **c*)

Private Types

typedef *hpx*::*util*::iterator_facade<channel_async_iterator<T>, *hpx*::future<T>, *std*::input_iterator_tag, *hpx*::fut

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_itera
#include <channel.hpp>

Public Functions

channel_iterator()

channel_iterator(detail::channel_base<T> **const** **c*)

channel_iterator(receive_channel<T> **const** **c*)

Private Types

typedef *hpx*::*util*::iterator_facade<channel_iterator<T>, T **const**, *std*::input_iterator_tag> **base_type**

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_typ*
#include <channel.hpp>

Public Functions

channel_iterator()

channel_iterator(detail::channel_base<void> **const** **c*)

channel_iterator(*receive_channel*<void> **const** **c*)

Private Types

typedef *hpx*::*util*::iterator_facade<channel_iterator<void>, *util*::*unused_type* **const**, *std*::input_iterator_tag> **b**

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 : **protected** *hpx*::*lcos*::*local*::detail::channel_base<*T*>
#include <channel.hpp>

Public Types

typedef T **value_type**

Public Functions

one_element_channel()

Private Types

typedef detail::channel_base<T> **base_type**

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>
#include <channel.hpp>

Public Types

typedef void **value_type**

Public Functions

one_element_channel()

Private Types

typedef detail::channel_base<void> **base_type**

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 : **protected** *hpx*::*lcos*::*local*::detail::channel_base<*T*>
#include <channel.hpp>

Public Functions

receive_channel(channel<T> **const** &*c*)

receive_channel(one_element_channel<T> **const** &*c*)

Private Types

typedef detail::channel_base<T> **base_type**

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>
#include <channel.hpp>

Public Functions

receive_channel (*channel*<void> **const** &*c*)

receive_channel (*one_element_channel*<void> **const** &*c*)

Private Types

typedef detail::channel_base<void> **base_type**

Friends

friend hpx::lcos::local::channel_iterator< void >

friend hpx::lcos::local::send_channel< void >

template<typename **T**>
class send_channel : **private** *hpx*::*lcos*::*local*::detail::channel_base<*T*>
#include <channel.hpp>

Public Functions

send_channel (channel<T> **const** &*c*)

send_channel (one_element_channel<T> **const** &*c*)

Private Types

typedef detail::channel_base<T> **base_type**

template<>
class send_channel<void> : **private** *hpx*::*lcos*::*local*::detail::channel_base<void>
#include <channel.hpp>

Public Functions

send_channel(*channel*<void> **const** &*c*)

send_channel(*one_element_channel*<void> **const** &*c*)

Private Types

typedef detail::channel_base<void> **base_type**

Header `hpx/lcos_local/composable_guard.hpp`

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
#include <composable_guard.hpp>

Public Functions

guard()

~guard()

Public Members

detail::guard_atomic **task**

class guard_set : **public** *hpx*::*lcos*::*local*::detail::debug_object
#include <composable_guard.hpp>

Public Functions

guard_set()

~guard_set()

std::shared_ptr<*guard*> **get**(*std*::size_t *i*)

void **add**(*std*::shared_ptr<*guard*> **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.

Header `hpx/lcos_local/conditional_trigger.hpp`

namespace hpx

namespace lcos

namespace local

struct conditional_trigger
#include <conditional_trigger.hpp>

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_**

Header `hpx/lcos_local/packaged_task.hpp`

namespace hpx

namespace lcos

namespace local

template<typename **R**, typename ...**Ts**>
class packaged_task<*R*(*Ts*...)>
#include <packaged_task.hpp>

Public Functions

packaged_task()

template<typename **F**, typename **FD** = **typename** *std*::decay<*F*>::type, typename **Enable** = **typename** *std*::en
packaged_task(*F* &&*f*)

template<typename **Allocator**, typename **F**, typename **FD** = **typename** *std*::decay<*F*>::type, typename **Enab**
packaged_task(*std*::allocator_arg_t, *Allocator* **const** &*a*, *F* &&*f*)

packaged_task(*packaged_task* &&*rhs*)

packaged_task &**operator=**(packaged_task &&*rhs*)

void **swap**(packaged_task &*rhs*)

void **operator()**(Ts... *vs*)

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

typedef *util*::*unique_function_nonser*<R (Ts...) > **function_type**

Private Functions

template<typename ...**Vs**>
void **invoke_impl** (*std*::false_type, *Vs*&&... *vs*)

template<typename ...**Vs**>
void **invoke_impl** (*std*::true_type, *Vs*&&... *vs*)

Private Members

function_type **function_**

local::promise<R> **promise_**

Header `hpx/lcos_local/promise.hpp`

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*>
#include <promise.hpp>

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*)

bool **valid**() **const**

void **set_value**(R **const** &*r*)

void **set_value**(R &&*r*)

template<typename ...**Ts**>
void **set_value**(*Ts*&&... *ts*)

void **set_exception**(*std*::exception_ptr *e*)

Private Types

typedef detail::promise_base<R> **base_type**

template<typename **R**>
class promise<*R*&> : **public** *hpx*::*lcos*::*local*::detail::promise_base<*R*&>
#include <promise.hpp>

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*)

bool **valid**() **const**

void **set_value**(R &*r*)

void **set_exception**(*std*::exception_ptr *e*)

Private Types

typedef detail::promise_base<R&> **base_type**

template<>
class promise<void> : **public** *hpx*::*lcos*::*local*::detail::promise_base<void>
#include <promise.hpp>

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*)

bool **valid**() **const**

void **set_value**()

void **set_exception**(*std*::exception_ptr *e*)

Private Types

typedef detail::promise_base<void> **base_type**

Header `hpx/lcos_local/receive_buffer.hpp`

namespace hpx

namespace lcos

namespace local

template<typename **T**, typename **Mutex** = *lcos*::*local*::*spinlock*>
struct receive_buffer
#include <receive_buffer.hpp>

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
#include <receive_buffer.hpp>

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
#include <receive_buffer.hpp>

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<typename **Mutex**>
struct receive_buffer<void, *Mutex*>
#include <receive_buffer.hpp>

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<typename **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
#include <receive_buffer.hpp>

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
#include <receive_buffer.hpp>

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_**

Header `hpx/lcos_local/trigger.hpp`

namespace hpx

namespace lcos

namespace local

template<typename **Mutex** = *lcos::local::spinlock*>
struct base_trigger
#include <trigger.hpp>

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* = "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 **Lock**>
void **synchronize**(*std*::size_t *generation_value*, *Lock* *&l*, char **const** **function_name* = "base_and_gate<>::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*>
#include <trigger.hpp>

Public Functions

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

typedef base_trigger<*no_mutex*> **base_type**

Header `hpx/local/channel.hpp`

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.

Header `hpx/logging/format/destinations.hpp`

namespace hpx

namespace util

namespace logging

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()

struct file : **public** *hpx*::*util*::*logging*::*destination*::*manipulator*
#include <destinations.hpp> Writes the string to a file.

Public Functions

~file()

Public Static Functions

static *std*::unique_ptr<*file*> **make**(*std*::string **const** &*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

file(*std*::string **const** &*file_name*, *file_settings set*)

Protected Attributes

std::string **name**

file_settings **settings**

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

file_settings()

Public Members

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**

Header `hpx/logging/format/formatters.hpp`

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 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:

```
high_precision_time("$mm:$ss:$micro");
```

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

~high_precision_time()

Public Static Functions

static *std*::unique_ptr<*high_precision_time*> **make**(*std*::string **const** &*format*)

Protected Functions

high_precision_time(*std*::string **const** &*format*)

struct idx : **public** *hpx*::*util*::*logging*::*formatter*::*manipulator*

#include <formatters.hpp> prefixes each message with an index.

Example:

```
L_ << "my message";
L_ << "my 2nd message";
```

This will output something similar to:

```
[1] my message
[2] my 2nd message
```

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()

Header hpx/logging/format/named_write.hpp

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.

```
#include <hpx/logging/format/named_write.hpp>
```

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:

```
// Set the formatters (first param) and destinatins (second step) in one step
g_l()->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()

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

void **destination**(*std*::string **const** &*destination_str*)
sets the destinations string - where should logged messages be outputted

void **write**(*std*::string **const** &*format_str*, *std*::string **const** &*destination_str*)
Specifies the formats and destinations in one step.

void **operator()**(*message* **const** &*msg*) **const**

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.

template<typename **Formatter**, typename ...**Args**>
void **set_formatter**(*std*::string **const** &*name*, *Args&&... args*)

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.

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**

Header hpx/logging/level.hpp

namespace hpx

namespace util

namespace logging

Enums

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:

disable_all = static_cast<unsigned int>(-1)

enable_all = 0

debug = 1000

info = 2000

warning = 3000

error = 4000

fatal = 5000

always = 6000

Functions

void **format_value**(*std*::ostream &*os*, *boost*::string_ref *spec*, *level value*)

Header `hpx/logging/logging.hpp`

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:

```
#include <hpx/logging/format.hpp>
```

Header `hpx/logging/manipulator.hpp`

namespace hpx

namespace util

namespace logging

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 manipulator

#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

virtual void **operator()**(*message* **const** &*val*) = 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()

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 &*to*) **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*)

Header `hpx/logging/message.hpp`

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.

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*)

Header `hpx/modules/logging.hpp`

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

#include <logging.hpp>

Public Functions

constexpr bootstrap_logging ()

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.

Header `hpx/memory/intrusive_ptr.hpp`

template<typename **T**>
struct hash<*hpx*::*memory*::intrusive_ptr<*T*>>
#include <intrusive_ptr.hpp>

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
#include <intrusive_ptr.hpp>

Public Types

template<>
using element_type = T

Public Functions

constexpr intrusive_ptr ()

intrusive_ptr (T **p*, bool *add_ref* = true)

template<typename **U**, typename **Enable** = **typename** *std*::enable_if<*memory*::detail::sp_convertible<*U*, T>::value>
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*)

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>
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*)

void **reset**()

void **reset**(T **rhs*)

void **reset**(T **rhs*, bool *add_ref*)

T ***get**() **const**

T ***detach**()

T &**operator***() **const**

T ***operator->**() **const**

operator bool() **const**

void **swap**(intrusive_ptr &*rhs*)

Private Types

template<>
using this_type = intrusive_ptr

Private Members

T ***px**

Friends

friend hpx::memory::intrusive_ptr

namespace std

template<typename **T**>
struct hash<*hpx*::*memory*::intrusive_ptr<*T*>>
#include <intrusive_ptr.hpp>

Public Types

template<>
using result_type = *std*::size_t

Public Functions

result_type **operator()** (*hpx*::*memory*::intrusive_ptr<T> **const** &*p*) **const**

Header `hpx/memory/serialization/intrusive_ptr.hpp`

namespace hpx

namespace serialization

Functions

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)

hpx::serialization::HPX_SERIALIZATION_SPLIT_FREE_TEMPLATE((template< typename T >),

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.

Header `hpx/mpi_base/mpi.hpp`

Header `hpx/mpi_base/mpi_environment.hpp`

namespace hpx

namespace util

struct mpi_environment
#include <mpi_environment.hpp>

Public Static Functions

static bool **check_mpi_environment** (*runtime_configuration* **const** &*cfg*)

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.

Header `hpx/modules/naming_base.hpp`

namespace hpx

namespace naming

Typedefs

using component_type = *std*::int32_t

using address_type = *std*::uint64_t

Variables

constexpr *std*::uint32_t **invalid_locality_id** = ~static_cast<*std*::uint32_t>(0)

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.

Header `hpx/pack_traversal/pack_traversal.hpp`

namespace hpx

namespace util

Functions

template<typename Mapper, typename... T><unspecified> hpx::util::map_pack(Mapper &&

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::util::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.

Header hpx/pack_traversal/pack_traversal_async.hpp

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)
    {
```

(continues on next page)

(continued from previous page)

```
    }

    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.

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:

```
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.

Header `hpx/pack_traversal/traits/pack_traversal_rebind_container.hpp`

template<typename **NewType**, typename **OldType**, typename **OldAllocator**>
struct pack_traversal_rebind_container<*NewType*, *std*::vector<*OldType*, *OldAllocator*>>
#include <pack_traversal_rebind_container.hpp>

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*)

template<typename **NewType**, typename **OldType**, typename **OldAllocator**>
struct pack_traversal_rebind_container<*NewType*, *std*::list<*OldType*, *OldAllocator*>>
#include <pack_traversal_rebind_container.hpp>

Public Types

template<>
using NewAllocator = **typename** *std*::allocator_traits<OldAllocator>::template rebind_alloc<NewType>

Public Static Functions

static *std*::list<NewType, NewAllocator> **call** (*std*::list<OldType, OldAllocator> **const** *&container*)

template<typename **NewType**, typename **OldType**, *std*::size_t **N**>
struct pack_traversal_rebind_container<*NewType*, *std*::array<*OldType*, *N*>>
#include <pack_traversal_rebind_container.hpp>

Public Static Functions

static *std*::array<NewType, N> **call** (*std*::array<OldType, N> **const**&)

namespace hpx

namespace traits

template<typename **NewType**, typename **OldType**, *std*::size_t **N**>
struct pack_traversal_rebind_container<*NewType*, *std*::array<*OldType*, *N*>>
#include <pack_traversal_rebind_container.hpp>

Public Static Functions

static *std*::array<NewType, N> **call** (*std*::array<OldType, N> **const**&)

template<typename **NewType**, typename **OldType**, typename **OldAllocator**>
struct pack_traversal_rebind_container<*NewType*, *std*::list<*OldType*, *OldAllocator*>>
#include <pack_traversal_rebind_container.hpp>

Public Types

template<>
using NewAllocator = **typename** *std*::allocator_traits<OldAllocator>::template rebind_alloc<NewType>

Public Static Functions

static *std*::list<NewType, NewAllocator> **call** (*std*::list<OldType, OldAllocator> **const** *&container*)

template<typename **NewType**, typename **OldType**, typename **OldAllocator**>
struct pack_traversal_rebind_container<*NewType*, *std*::vector<*OldType*, *OldAllocator*>>
#include <pack_traversal_rebind_container.hpp>

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*)

Header `hpx/pack_traversal/unwrap.hpp`

namespace hpx

namespace util

Functions

template<typename ...**Args**>
auto **unwrap** (*Args&&... args*)
A helper function for retrieving the actual result of any hpx::lcos::future like type which is wrapped in an arbitrary way.

Unwraps the given pack of arguments, so that any hpx::lcos::future object is replaced by its future result type in the argument pack:
- `hpx::future<int>` -> `int`
- `hpx::future<std::vector<float>>` -> `std::vector<float>`

- `std::vector<future<float>>` -> `std::vector<float>`

The function is capable of unwrapping hpx::lcos::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::lcos::future like types are permitted as arguments and passed through.

```
// Single arguments
int i1 = hpx:util::unwrap(hpx::lcos::make_ready_future(0));

// Multiple arguments
hpx::tuple<int, int> i2 =
    hpx:util::unwrap(hpx::lcos::make_ready_future(1),
                     hpx::lcos::make_ready_future(2));
```

Note This function unwraps the given arguments until the first traversed nested hpx::lcos::future which corresponds to an unwrapping depth of one. See *hpx::util::unwrap_n()* for a function which unwraps the given arguments to a particular depth or *hpx::util::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::util::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::lcos::future objects were resolved through an exception. See hpx::lcos::future::get() for details.

template<*std*::size_t **Depth**, typename ...**Args**>
auto **unwrap_n**(*Args&&... args*)

An alterntive version of *hpx::util::unwrap()*, which unwraps the given arguments to a certain depth of hpx::lcos::future like objects.

See unwrap for a detailed description.

Template Parameters

- `Depth`: The count of hpx::lcos::future like objects which are unwrapped maximally.

template<typename ...**Args**>
auto **unwrap_all**(*Args&&... args*)

An alterntive version of *hpx::util::unwrap()*, which unwraps the given arguments recursively so that all contained hpx::lcos::future like objects are replaced by their actual value.

See *hpx::util::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::util::unwrap()* function and then passes the result to the given callable object.

```
auto callable = hpx::util::unwrapping([](int left, int right) {
    return left + right;
});

int i1 = callable(hpx::lcos::make_ready_future(1),
                  hpx::lcos::make_ready_future(2));
```

See *hpx::util::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::util::unwrap_n()* function and then passes the result to the given callable object.

See *hpx::util::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::util::unwrap_all()* function and then passes the result to the given callable object.

See *hpx::util::unwrapping()* for a detailed description.

namespace functional

struct unwrap

#include <unwrap.hpp> A helper function object for functionally invoking *hpx::util::unwrap*. For more information please refer to its documentation.

struct unwrap_all

#include <unwrap.hpp> A helper function object for functionally invoking *hpx::util::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::util::unwrap_n*. For more information please refer to its documentation.

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.

Header hpx/performance_counters/apex_sample_value.hpp

Header hpx/performance_counters/base_performance_counter.hpp

namespace hpx

namespace performance_counters

template<typename **Derived**>
class base_performance_counter : **public** *hpx*::*performance_counters*::*server*::*base_performance_counter*, **p**

#include <base_performance_counter.hpp>

Public Types

typedef Derived **type_holder**

typedef *hpx*::*performance_counters*::*server*::*base_performance_counter* **base_type_holder**

Public Functions

base_performance_counter()

base_performance_counter(*hpx*::*performance_counters*::*counter_info* **const** &*info*)

void **finalize**()

Private Types

typedef *hpx*::*components*::component_base<Derived> **base_type**

Header `hpx/performance_counters/counter_creators.hpp`

namespace hpx

namespace performance_counters

Functions

bool **default_counter_discoverer**(*counter_info* **const**&, *discover_counter_func* **const**&, *discover_counters_mode*, *error_code*&)

Default discovery function for performance counters; to be registered with the counter types. It will pass the counter_info and the *error_code* to the supplied function.

bool **locality_counter_discoverer**(*counter_info* **const**&, *discover_counter_func* **const**&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/total)/<instancename>

bool **locality_pool_counter_discoverer**(*counter_info* **const**&, *discover_counter_func* **const**&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/pool#<pool_name>/total)/<instancename>

bool **locality0_counter_discoverer**(*counter_info* **const**&, *discover_counter_func* **const**&, *discover_counters_mode*, *error_code*&)

Default discoverer function for AGAS performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#0/total}/<instancename>

bool **locality_thread_counter_discoverer**(*counter_info* **const**&, *discover_counter_func* **const**&, *discover_counters_mode*, *error_code*&)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>(locality#<locality_id>/worker-thread#<threadnum>)/<instancename>

bool **locality_pool_thread_counter_discoverer**(*counter_info* **const** &*info*, *discover_counter_func* **const** &*f*, *discover_counters_mode* *mode*, *error_code* &*ec*)

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#<locality_id>/pool#<poolname>/thread#<threadnum>}/<instancename>

total is not* supported* bool hpx::performance_counters::locality_pool_thread_no_to

Default discoverer function for performance counters; to be registered with the counter types. It is suitable to be used for all counters following the naming scheme:

/<objectname>{locality#<locality_id>/pool#<poolname>/thread#<threadnum>}/<instancename>

This is essentially the same as above just that locality#

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>

> **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*&)

Header hpx/performance_counters/counter_parser.hpp

namespace hpx

namespace performance_counters

Functions

bool **parse_counter_name**(*std*::string **const** &*name*, *path_elements* &*elements*)

struct instance_elements

#include <counter_parser.hpp>

Public Members

instance_name **parent_**

instance_name **child_**

instance_name **subchild_**

struct instance_name

#include <counter_parser.hpp>

Public Members

std::string **name_**

std::string **index_**

bool **basename_** = false

struct path_elements

#include <counter_parser.hpp>

Public Members

std::string **object_**

instance_elements **instance_**

std::string **counter_**

std::string **parameters_**

Header `hpx/performance_counters/counters.hpp`

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: (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.

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

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

std::string &**ensure_counter_prefix**(*std*::string &*name*)

std::string **ensure_counter_prefix**(*std*::string **const** &*counter*)

std::string &**remove_counter_prefix**(*std*::string &*name*)

std::string **remove_counter_prefix**(*std*::string **const** &*counter*)

char **const** ***get_counter_type_name**(*counter_type state*)

Return the readable name of a given counter type.

bool **status_is_valid**(*counter_status s*)

counter_status **add_counter_type**(*counter_info* **const** &*info*, *error_code* &*ec*)

naming::id_type **get_counter**(*std*::string **const** &*name*, *error_code* &*ec*)

naming::id_type **get_counter**(*counter_info* **const** &*info*, *error_code* &*ec*)

Variables

constexpr char **const** **counter_prefix**[] = "/counters"

constexpr *std*::size_t **counter_prefix_len** = (sizeof(*counter_prefix*) / sizeof(*counter_prefix*[0])) - 1

struct counter_info
#include <counters.hpp>

Public Functions

counter_info(*counter_type type* = *counter_raw*)

counter_info(*std*::string **const** &*name*)

counter_info(*counter_type type*, *std*::string **const** &*name*, *std*::string **const** &*helptext* = "", *std*::uint32_t *version* = HPX_PERFORMANCE_COUNTER_V1, *std*::string **const** &*uom* = "")

Public Members

counter_type **type_**
The type of the described counter.

std::uint32_t **version_**
The version of the described counter using the 0xMMmmSSSS scheme

counter_status **status_**
The status of the counter object.

std::string **fullname_**
The full name of this counter.

std::string **helptext_**
The full descriptive text for this counter.

std::string **unit_of_measure_**
The unit of measure for this counter.

Private Functions

void **serialize**(*serialization*::*output_archive* &*ar*, **const** unsigned int)

void **serialize**(*serialization*::*input_archive* &*ar*, **const** unsigned int)

Friends

friend hpx::performance_counters::hpx::serialization::access

struct counter_path_elements : **public** *hpx*::*performance_counters*::*counter_type_path_elements*
#include <counters.hpp> A *counter_path_elements* holds the elements of a full name for a counter instance. Generally, a full name of a counter instance has the structure:

/objectname{parentinstancename::parentindex/instancename#instanceindex} /countername#parameters

i.e. /queue{localityprefix/thread#2}/length

Public Types

typedef *counter_type_path_elements* **base_type**

Public Functions

counter_path_elements()

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

Header `hpx/performance_counters/counters_fwd.hpp`

Defines

HPX_PERFORMANCE_COUNTER_V1

namespace hpx

namespace performance_counters

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: (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.

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_mode* *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*) *counter_status* **discover_counter_type**(*std*::string **const** &*name*, *std*::vector<*counter_info*> &*counters*, *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*&)

call the supplied function will all expanded versions of the supplied counter info.

This function expands all locality#* and worker-thread#* wild cards only.

counter_status **remove_counter_type**(*counter_info* **const** &*info*, *error_code* &*ec* = *throws*)

Remove an existing counter type from the (local) registry.

Note This doesn't remove existing counters of this type, it just inhibits defining new counters using this type.

counter_status **get_counter_type**(*std*::string **const** &*name*, *counter_info* &*info*, *error_code* &*ec* = *throws*)

Retrieve the counter type for the given counter name from the (local) registry.

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 &*helptext*, *std*::uint32_t &*version*, *error_code* &*ec* = *throws*)

Retrieve the meta data specific for the given counter instance.

void **get_counter_infos**(*std*::string *name*, *counter_type* &*type*, *std*::string &*helptext*, *std*::uint32_t &*version*, *error_code* &*ec* = *throws*)

Retrieve the meta data specific for the given counter instance.

struct counter_value

#include <counters_fwd.hpp>

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
#include <counters_fwd.hpp>

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)

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

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

Header `hpx/performance_counters/manage_counter.hpp`

namespace hpx

namespace performance_counters

Functions

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.

Header `hpx/performance_counters/manage_counter_type.hpp`

namespace hpx

namespace performance_counters

Functions

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 provided function to be called and the returned array value to be exposed as the counter value.

The counter type is registered such that there can be one counter instance per locality. The expected naming scheme for the counter instances is: `'/objectname{locality#<*>/total}/countername'` where '<*>' is a zero based integer identifying the locality the counter is created on.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

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 (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 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.
- `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*, *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.
- `helptext`: [in] A longer descriptive text shown to the user to explain the nature of the counters created from this type.
- `version`: [in] The version of the counter type. This is currently expected to be set to HPX_PERFORMANCE_COUNTER_V1.
- `create_counter`: [in] The function which will be called to create a new instance of this counter type.
- `discover_counters`: [in] The function will be called to discover counter instances which can be created.
- `uom`: [in] The unit of measure of the counter type (default: "")
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Header `hpx/performance_counters/parcels/data_point.hpp`

`namespace hpx`

`namespace performance_counters`

`namespace parcels`

struct data_point

#include <data_point.hpp> A data_point collects all timing and statistical information for a single parcel (either sent or received).

Public Functions

data_point()

Public Members

std::size_t **bytes_**
number of bytes on the wire for this parcel (possibly compressed)

std::int64_t **time_**
during processing holds start timestamp after processing holds elapsed time

std::int64_t **serialization_time_**
during processing holds start serialization timestamp after processing holds elapsed serialization time

std::size_t **num_parcels_**
The number of parcels processed by this message.

std::size_t **raw_bytes_**
number of bytes processed for the action in this parcel (uncompressed)

std::int64_t **buffer_allocate_time_**
The time spent for allocating buffers.

Header `hpx/performance_counters/parcels/gatherer.hpp`

namespace hpx

namespace performance_counters

namespace parcels

Typedefs

using gatherer = detail::gatherer<*lcos*::*local*::*spinlock*>

using gatherer_nolock = detail::gatherer<*lcos*::*local*::*no_mutex*>

Header hpx/performance_counters/performance_counter.hpp

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_performan*

#include <performance_counter.hpp>

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*)

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*)

Header `hpx/performance_counters/performance_counter_base.hpp`

namespace hpx

namespace performance_counters

struct performance_counter_base
#include <performance_counter_base.hpp> Subclassed by *hpx::performance_counters::server::base_performance_co*

Public Functions

virtual ~performance_counter_base()
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**(*counter_value* **const**&) = 0

virtual bool **start**() = 0

virtual bool **stop**() = 0

virtual void **reinit**(bool *reset*) = 0

Header `hpx/performance_counters/performance_counter_set.hpp`

namespace hpx

namespace performance_counters

class performance_counter_set
#include <performance_counter_set.hpp>

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_**

Header hpx/performance_counters/registry.hpp

namespace hpx

namespace performance_counters

class registry
#include <registry.hpp>

Public Functions

registry()

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.

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_**

Header hpx/performance_counters/server/arithmetics_counter.hpp

namespace hpx

namespace performance_counters

namespace server

template<typename **Operation**>
class arithmetics_counter : **public** *hpx*::*performance_counters*::*server*::*base_performance_counter*, **pub**
#include <arithmetics_counter.hpp>

Public Types

typedef arithmetics_counter **type_holder**

typedef *base_performance_counter* **base_type_holder**

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**()
the following functions are not implemented by default, they will just throw

void **finalize**()

Private Types

typedef *components*::component_base<arithmetics_counter<Operation>> **base_type**

Private Members

performance_counter_set **counters_**

Header `hpx/performance_counters/server/arithmetics_counter_extended.hpp`

namespace hpx

namespace performance_counters

namespace server

template<typename **Statistic**>
class arithmetics_counter_extended : **public** *hpx*::*performance_counters*::*server*::*base_performance_*
#include <arithmetics_counter_extended.hpp>

Public Types

typedef arithmetics_counter_extended **type_holder**

typedef *base_performance_counter* **base_type_holder**

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**()
the following functions are not implemented by default, they will just throw

void **finalize**()

Private Types

typedef *components*::component_base<arithmetics_counter_extended<Statistic>> **base_type**

Private Members

performance_counter_set **counters_**

Header `hpx/performance_counters/server/base_performance_counter.hpp`

namespace hpx

namespace performance_counters

namespace server

class base_performance_counter : **public** *hpx*::*performance_counters*::*performance_counter_base*, **pub**
#include <base_performance_counter.hpp> Subclassed by *hpx::performance_counters::base_performance_counte* *Derived* >, *hpx::performance_counters::server::arithmetics_counter< Operation* >, *hpx::performance_counters::server::arithmetics_counter_extended< Statistic* >, *hpx::performance_counters::server::elapsed_time_counter*, *hpx::performance_counters::server::raw_counter*, *hpx::performance_counters::server::raw_values_counter*, *hpx::performance_counters::server::statistics_counter< Statistic >*

Public Types

typedef *components*::component<*base_performance_counter*> **wrapping_type**

typedef *base_performance_counter* **base_type_holder**

Public Functions

base_performance_counter()

base_performance_counter(*counter_info* **const** &*info*)

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*)

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.

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.

HPX_DEFINE_COMPONENT_ACTION (*base_performance_counter*, *get_counter_values_array_nonvirt*, get_counter_values_array_action)

The *get_counter_value_action* queries the value of a performance counter.

HPX_DEFINE_COMPONENT_ACTION (*base_performance_counter*, *set_counter_value_nonvirt*, set_counter_value_action)

The *set_counter_value_action*.

HPX_DEFINE_COMPONENT_ACTION (*base_performance_counter*, *reset_counter_value_nonvirt*, reset_counter_value_action)

The *reset_counter_value_action*.

HPX_DEFINE_COMPONENT_ACTION (*base_performance_counter*, *start_nonvirt*, start_action)

The *start_action*.

HPX_DEFINE_COMPONENT_ACTION (*base_performance_counter*, *stop_nonvirt*, stop_action)

The *stop_action*.

HPX_DEFINE_COMPONENT_ACTION (*base_performance_counter*, *reinit_nonvirt*, 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

virtual void **reset_counter_value**()
the following functions are not implemented by default, they will just throw

virtual void **set_counter_value**(*counter_value* **const**&)

virtual *counter_value* **get_counter_value**(bool)

virtual *counter_values_array* **get_counter_values_array**(bool)

virtual bool **start**()

virtual bool **stop**()

virtual void **reinit**(bool)

virtual *counter_info* **get_counter_info**() **const**

Protected Attributes

hpx::*performance_counters*::*counter_info* **info_**

util::*atomic_count* **invocation_count_**

Header `hpx/performance_counters/server/elapsed_time_counter.hpp`

namespace hpx

namespace performance_counters

namespace server

class elapsed_time_counter : **public** *hpx*::*performance_counters*::*server*::*base_performance_counter*, **pu**
#include <elapsed_time_counter.hpp>

Public Types

typedef *elapsed_time_counter* **type_holder**

typedef *base_performance_counter* **base_type_holder**

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**()
finalize() will be called just before the instance gets destructed

Private Types

typedef *components*::component_base<*elapsed_time_counter*> **base_type**

Header `hpx/performance_counters/server/raw_counter.hpp`

namespace hpx

namespace performance_counters

namespace server

class raw_counter : **public** *hpx*::*performance_counters*::*server*::*base_performance_counter*, **public** *hpx*::*co*
#include <raw_counter.hpp>

Public Types

typedef *raw_counter* **type_holder**

typedef *base_performance_counter* **base_type_holder**

Public Functions

raw_counter()

raw_counter(*counter_info* **const** &*info*, *hpx*::*util*::*function_nonser*<*std*::int64_t) bool
> *f*

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**()
finalize() will be called just before the instance gets destructed

Private Types

typedef *components*::component_base<*raw_counter*> **base_type**

Private Members

hpx::*util*::*function_nonser*<*std*::int64_t (bool) > **f_**

bool **reset_**

Header `hpx/performance_counters/server/raw_values_counter.hpp`

namespace hpx

namespace performance_counters

namespace server

class raw_values_counter : **public** *hpx*::*performance_counters*::*server*::*base_performance_counter*, **publ**

#include <raw_values_counter.hpp>

Public Types

typedef *raw_values_counter* **type_holder**

typedef *base_performance_counter* **base_type_holder**

Public Functions

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** ()

finalize() will be called just before the instance gets destructed

Private Types

typedef *components*::component_base<*raw_values_counter*> **base_type**

Private Members

hpx::util::function_nonser<std::vector<std::int64_t>bool)> hpx::performance_c

bool **reset_**

Header hpx/performance_counters/server/statistics_counter.hpp

namespace hpx

namespace performance_counters

namespace server

template<typename **Statistic**>
class statistics_counter : **public** *hpx*::*performance_counters*::*server*::*base_performance_counter*, **publ**

#include <statistics_counter.hpp>

Public Types

typedef statistics_counter **type_holder**

typedef *base_performance_counter* **base_type_holder**

Public Functions

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*)

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**()

the following functions are not implemented by default, they will just throw

void **on_terminate**()

void **finalize**()

finalize() will be called just before the instance gets destructed

Protected Functions

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_**
base time interval in milliseconds

std::string **base_counter_name_**
name of base counter to be queried

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_**

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.

Header hpx/modules/plugin.hpp

Header hpx/plugin/abstract_factory.hpp

Header hpx/plugin/concrete_factory.hpp

Header hpx/plugin/config.hpp

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

namespace hpx

namespace util

namespace plugin

Typedefs

using shared_ptr = *boost*::shared_ptr<T>

Header hpx/plugin/dll.hpp

Defines

HPX_HAS_DLOPEN

Header hpx/plugin/export_plugin.hpp

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*)

Header `hpx/plugin/plugin_factory.hpp`

namespace hpx

namespace util

namespace plugin

template<class **BasePlugin**>
struct plugin_factory : **public** *hpx*::*util*::*plugin*::detail::plugin_factory_item<*BasePlugin*, detail::plugin_fa
#include <plugin_factory.hpp>

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** v

template<class **BasePlugin**>
struct static_plugin_factory : **public** *hpx*::*util*::*plugin*::detail::static_plugin_factory_item<*BasePlugin*,
#include <plugin_factory.hpp>

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, t

Header `hpx/plugin/plugin_wrapper.hpp`

namespace hpx

namespace util

namespace plugin

template<typename **Wrapped**, typename ...**Parameters**>
struct plugin_wrapper : **public** *hpx*::*util*::*plugin*::detail::dll_handle_holder, **public** *Wrapped*
#include <plugin_wrapper.hpp>

Public Functions

plugin_wrapper (*dll_handle dll*, Parameters... *parameters*)

Header `hpx/plugin/traits/plugin_config_data.hpp`

namespace hpx

namespace traits

template<typename **Plugin**, typename **Enable** = void>
struct plugin_config_data
#include <plugin_config_data.hpp>

Public Static Functions

static char **const** ***call** ()

Header hpx/plugin/virtual_constructor.hpp

namespace hpx

namespace util

namespace plugin

Typedefs

using exported_plugins_type = *std*::map<*std*::string, *hpx*::*util*::*any_nonser*>

typedef exported_plugins_type*(HPX_PLUGIN_API* hpx::util::plugin::get_plugins_li

typedef exported_plugins_type* HPX_PLUGIN_API hpx::util::plugin::get_plugins_lis

using dll_handle = *shared_ptr*<get_plugins_list_np>

template<typename **BasePlugin**>
struct virtual_constructor
#include <virtual_constructor.hpp>

Public Types

template<>
using type = *hpx*::*util*::pack<>

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.

Header hpx/prefix/find_prefix.hpp

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)

preprocessor

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.

Header `hpx/preprocessor/cat.hpp`

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

Header `hpx/preprocessor/expand.hpp`

Defines

HPX_PP_EXPAND(*X*)
The HPX_PP_EXPAND macro performs a double macro-expansion on its argument. This macro can be used to produce a delayed preprocessor expansion.

Parameters

- `X`: Token to be expanded twice

Example:

```
#define MACRO(a, b, c) (a)(b)(c)
#define ARGS() (1, 2, 3)

HPX_PP_EXPAND(MACRO ARGS()) // expands to (1)(2)(3)
```

Header hpx/preprocessor/nargs.hpp

Defines

HPX_PP_NARGS(...)

Expands to the number of arguments passed in

Example Usage:

```
HPX_PP_NARGS(hpx, pp, nargs)
HPX_PP_NARGS(hpx, pp)
HPX_PP_NARGS(hpx)
```

Parameters

- `...`: The variadic number of arguments

Expands to:

```
3
2
1
```

Header hpx/preprocessor/stringize.hpp

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

Header hpx/preprocessor/strip_parens.hpp

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.

Header `hpx/modules/program_options.hpp`

Header `hpx/program_options.hpp`

Header `hpx/program_options/cmdline.hpp`

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*
The more-or-less traditional unix style.

default_style = *unix_style*
The default style.

Header `hpx/program_options/config.hpp`

namespace hpx

namespace program_options

Typedefs

using any = *hpx*::*util*::*any_nonser*

using optional = *hpx*::*util*::optional<T>

Header `hpx/program_options/environment_iterator.hpp`

namespace hpx

namespace program_options

class environment_iterator : **public** *hpx*::*program_options*::eof_iterator<*environment_iterator*, *std*::pair<*std*::
#include <environment_iterator.hpp>

Public Functions

environment_iterator (char ***environment*)

environment_iterator ()

void **get** ()

Private Members

char ****m_environment**

Header `hpx/program_options/eof_iterator.hpp`

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

Header `hpx/program_options/errors.hpp`

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> &*xalternatives*)

~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::error_with_no_option_name*, *hpx::program_options::invalid_syntax*, *hpx::program_options::multiple_occurrences*, *hpx::program_options::multiple_values*, *hpx::program_options::required_option*, *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 &*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*
#include <errors.hpp>

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

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*
#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 *hpx::program_options::invalid_command_line_syntax*, *hpx::program_options::invalid_config_file_syntax*

Public Types

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

invalid_syntax(*kind_t kind*, **const** *std*::string &*option_name* = "", **const** *std*::string &*original_token* = "", int *option_style* = 0)

~invalid_syntax()

kind_t **kind**() **const**

virtual *std*::string **tokens**() **const**
Convenience functions for backwards compatibility

Protected Functions

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

too_many_positional_options_error()

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

unknown_option(**const** *std*::string &*original_token* = "")

~unknown_option()

class validation_error : **public** *hpx*::*program_options*::*error_with_option_name*
#include <errors.hpp> Class thrown when value of option is incorrect.

Subclassed by *hpx::program_options::invalid_bool_value*, *hpx::program_options::invalid_option_value*

Public Types

enum kind_t
Values:

multiple_values_not_allowed = 30

at_least_one_value_required

invalid_bool_value

invalid_option_value

invalid_option

Public Functions

validation_error(*kind_t kind*, **const** *std*::string &*option_name* = "", **const** *std*::string &*original_token* = "", int *option_style* = 0)

~validation_error()

kind_t **kind**() **const**

Protected Functions

std::string **get_template**(*kind_t kind*)
Used to convert kind_t to a related error text

Protected Attributes

kind_t **m_kind**

Header `hpx/program_options/option.hpp`

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.

Header hpx/program_options/options_description.hpp

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 suitably 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

options_description(unsigned *line_length* = *m_default_line_length*, unsigned *min_description_length* = *m_default_line_length* / 2)
Creates the instance.

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.

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.

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.

std::size_t **get_option_column_width**() **const**
Find the maximum width of the option column, including options in groups.

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.

const *option_description* &**find**(**const** *std*::string &*name*, bool *approx*, bool *long_ignore_case* = false, bool *short_ignore_case* = false) **const**

const *option_description* ***find_nothrow**(**const** *std*::string &*name*, bool *approx*, bool *long_ignore_case* = false, bool *short_ignore_case* = false) **const**

const *std*::vector<*std*::shared_ptr<*option_description*>> &**options**() **const**

void **print**(*std*::ostream &*os*, *std*::size_t *width* = 0) **const**
Outputs 'desc' to the specified stream, calling 'f' to output each *option_description* element.

Public Static Attributes

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**

std::size_t **const m_line_length**

std::size_t **const m_min_description_length**

std::vector<*std*::shared_ptr<*option_description*>> **m_options**

std::vector<char> **belong_to_group**

std::vector<*std*::shared_ptr<*options_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**

Header `hpx/program_options/parsers.hpp`

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

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.

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.

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.

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 origianl 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*)
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 '

std::vector<*std*::wstring> **split_unix**(**const** *std*::wstring &*cmdline*)
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 basic_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 allow_long_disguise allow_dash_for_short allow_slash_for_short

template<>
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

Header hpx/program_options/positional_options.hpp

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**

Header `hpx/program_options/value_semantic.hpp`

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_codecvt_helper*<*Char*>, **public** *hpx*::*progra*
#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 explicit value is optional, but if given, must be strictly adjacent to the option, i.e.: '-ovalue' 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) **const** 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**

bool **is_composing**() **const**

unsigned **min_tokens**() **const**

unsigned **max_tokens**() **const**

bool **is_required**() **const**

void **xparse** (*hpx*::*util*::*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*::*util*::*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*::*util*::*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::*util*::*any_nonser* **m_default_value**

std::string **m_default_value_as_text**

hpx::*util*::*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

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>
#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

untyped_value(bool *zero_tokens* = false)

std::string **name**() **const**
Returns the name of the option. The name is only meaningful for automatic help message.

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_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.

bool **is_required**() **const**
Returns true if value must be given. Non-optional value

void **xparse**(*hpx*::*util*::*any_nonser* &*value_store*, **const** *std*::vector<*std*::string> &*new_tokens*) **const**
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.

bool **apply_default**(*hpx*::*util*::*any_nonser*&) **const**
Does nothing.

void **notify**(**const** *hpx*::*util*::*any_nonser*&) **const**
Does nothing.

Private Members

bool **m_zero_tokens**

class value_semantic
#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*::*util*::*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*::*util*::*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*::*util*::*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.

Subclassed by *hpx::program_options::typed_value*< *T, Char* >

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 input. 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::untyped_value*

Protected Functions

virtual void **xparse**(*hpx*::*util*::*any_nonser* &*value_store*, **const** *std*::vector<*std*::string> &*new_tokens*) **const** = 0

Private Functions

void **parse**(*hpx*::*util*::*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.

template<>
class value_semantic_codecvt_helper<wchar_t> : **public** *hpx*::*program_options*::*value_semantic*
#include <*value_semantic.hpp*> Helper conversion class for values that accept ascii strings as input. Overrides the 'parse' method and defines new 'xparse' method taking std::wstring. Depending on whether input to parse is ascii or UTF8, will recode input to Unicode, or pass it unmodified.

Protected Functions

virtual void **xparse**(*hpx*::*util*::*any_nonser* &*value_store*, **const** *std*::vector<*std*::wstring> &*new_tokens*) **const** = 0

Private Functions

void **parse**(*hpx*::*util*::*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.

Header `hpx/program_options/variables_map.hpp`

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*::*util*::*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.

template<class **T**>
T &**as** ()
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.

const *hpx*::*util*::*any_nonser* &**value** () **const**
Returns the contained value.

hpx::*util*::*any_nonser* &**value** ()
Returns the contained value.

Private Members

hpx::*util*::*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.

class variables_map : **public** *hpx*::*program_options*::*abstract_variables_map*, **public** *std*::map<*std*::string, *var*
#include <variables_map.hpp> Concrete variables map which store variables in real map.

This class is derived from std::map<std::string, variable_value>, so you can use all map operators to examine its content.

Public Functions

variables_map()

variables_map(**const** *abstract_variables_map* **next*)

const *variable_value* &**operator[]**(**const** *std*::string &*name*) **const**

void **clear**()

void **notify**()

Private Functions

const *variable_value* &**get**(**const** *std*::string &*name*) **const**
Implementation of abstract_variables_map::get which does 'find' in *this.

Private Members

std::set<*std*::string> **m_final**
Names of option with 'final' values – which should not be changed by subsequence assignments.

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

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.

Header `hpx/program_options/version.hpp`

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

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.

Header `hpx/resiliency/async_replay.hpp`

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_valid* *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*)

Header `hpx/resiliency/async_replay_executor.hpp`

namespace hpx

namespace resiliency

namespace experimental

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*)

template<typename **Executor**, typename **F**, typename ...**Ts**>
decltype(auto) **tag_invoke** (*async_replay_t*, *Executor* &&*exec*, *std*::size_t *n*, *F* &&*f*, *Ts*&&... *ts*)

Header `hpx/resiliency/async_replicate.hpp`

namespace hpx

namespace 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*, *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_vot* *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_val* *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_replicate_t*, *std*::size_t *n*, *F* &&*f*, *Ts*&&... *ts*)

Header `hpx/resiliency/async_replicate_executor.hpp`

namespace hpx

namespace resiliency

namespace experimental

Functions

template<typename **Executor**, typename **Vote**, typename **Pred**, typename **F**, typename ...**Ts**>
decltype(auto) **tag_invoke** (*async_replicate_vote_validate_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_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*)

Header hpx/resiliency/config.hpp

Header hpx/resiliency/replay_executor.hpp

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**>
replay_executor<*BaseExecutor*, detail::replay_validator> **make_replay_executor**(*BaseExecutor* &*exec*, *std*::size_t *n*)

template<typename **BaseExecutor**, typename **Validate**>
class replay_executor
#include <replay_executor.hpp>

Public Types

template<>
using execution_category = **typename** BaseExecutor::execution_category

template<>
using executor_parameters_type = **typename** BaseExecutor::executor_parameters_type

template<>
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

constexpr int **num_spread** = 4

constexpr int **num_tasks** = 128

Private Members

BaseExecutor &**exec_**

std::size_t **replay_count_**

Validate **validator_**

Header `hpx/resiliency/replicate_executor.hpp`

namespace hpx

namespace resiliency

namespace experimental

Functions

template<typename **BaseExecutor**, typename **Voter**, typename **Validate**>

replicate_executor<*BaseExecutor*, **typename** *std*::decay<*Voter*>::type, **typename** *std*::decay<*Validate*>::type> **mak**

template<typename **BaseExecutor**, typename **Validate**>
replicate_executor<*BaseExecutor*, detail::replicate_voter, **typename** *std*::decay<*Validate*>::type> **make_replicat**

template<typename **BaseExecutor**>
replicate_executor<*BaseExecutor*, detail::replicate_voter, detail::replicate_validator> **make_replicate_executo**

template<typename **BaseExecutor**, typename **Vote**, typename **Validate**>
class replicate_executor
#include <replicate_executor.hpp>

Public Types

template<>
using execution_category = **typename** BaseExecutor::execution_category

template<>
using executor_parameters_type = **typename** BaseExecutor::executor_parameters_type

template<>
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

constexpr int **num_spread** = 4

constexpr int **num_tasks** = 128

Private Members

BaseExecutor &**exec_**

std::size_t **replicate_count_**

Vote **voter_**

Validate **validator_**

Header hpx/resiliency/resiliency.hpp

Header hpx/resiliency/resiliency_cpos.hpp

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**

hpx::*resiliency*::*experimental*::*dataflow_replicate_vote_t* **dataflow_replicate_vote**

hpx::*resiliency*::*experimental*::*dataflow_replicate_validate_t* **dataflow_replicate_validate**

hpx::*resiliency*::*experimental*::*dataflow_replicate_t* **dataflow_replicate**

struct async_replay_t : **public** *hpx*::*functional*::*tag*<*async_replay_t*>

#include <*resiliency_cpos.hpp*> Customization point for asynchronously launching given function *f* repeatedly. Repeat launching on error exactly *n* times (except if abort_replay_exception is thrown).

struct async_replay_validate_t : **public** *hpx*::*functional*::*tag*<*async_replay_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 *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 *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 *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_validat*

#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 *pred*. Run all the valid results against a user provided voting function. Return the valid output.

struct dataflow_replay_t : **public** *hpx*::*resiliency*::*experimental*::*tag_deferred*<*dataflow_replay_t*, *async_r*

#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_replay_validate_t : **public** *hpx*::*resiliency*::*experimental*::*tag_deferred<dataflow_rep*

#include <resiliency_cpos.hpp> Customization point for asynchronously launching the given function *f*. repeatedly. Verify the result of those invocations using the given predicate *pred*. 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*,

#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*

#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 *pred*. Return the first valid result.

Delay the invocation of *f* if any of the arguments to *f* are futures.

struct dataflow_replicate_vote_t : **public** *hpx*::*resiliency*::*experimental*::*tag_deferred<dataflow_repli*

#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.

struct dataflow_replicate_vote_validate_t : **public** *hpx*::*resiliency*::*experimental*::*tag_deferred<d*

#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.

template<typename **Tag**, typename **BaseTag**>
struct tag_deferred : **public** *hpx*::*functional*::*tag<Tag>*

#include <resiliency_cpos.hpp>

Friends

template<typename ...**Args**>
auto **tag_invoke**(Tag, *Args&&... args*)

Header `hpx/resiliency/version.hpp`

Defines

HPX_RESILIENCY_VERSION_FULL

HPX_RESILIENCY_VERSION_MAJOR

HPX_RESILIENCY_VERSION_MINOR

HPX_RESILIENCY_VERSION_SUBMINOR

HPX_RESILIENCY_VERSION_DATE

namespace hpx

namespace resiliency

namespace experimental

Functions

unsigned int **major_version**()

unsigned int **minor_version**()

unsigned int **subminor_version**()

unsigned long **full_version**()

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.

Header `hpx/resource_partitioner/partitioner.hpp`

namespace hpx

namespace resource

class core

#include <partitioner.hpp>

Public Functions

core(*std*::size_t *id* = *invalid_core_id*, *numa_domain* **domain* = nullptr)

std::vector<*pu*> **const** &**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

#include <partitioner.hpp>

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

#include <partitioner.hpp>

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**

util::*command_line_handling* &**get_command_line_switches**()

void **configure_pools**()

int **parse_result**() **const**

Private Functions

partitioner (*util*::*function_nonser*<int> *hpx*::*program_options*::*variables_map* &vm > **const** &*f*, *hpx*::*program_options*::*options_description* **const** &*desc_cmdline*, int *argc*, char ***argv*, *std*::vector<*std*::string> *ini_config*, *resource*::*partitioner_mode rpmode*, *runtime_mode mode*, bool *check*, *std*::vector<*std*::shared_ptr<*components*::*component_registry_base*>> &*component_registries*, int **result*)

Private Members

detail::partitioner &**partitioner_**

class pu

#include <partitioner.hpp>

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

constexpr const *std*::size_t **invalid_pu_id** = *std*::size_t(-1)

Friends

friend hpx::resource::core

friend hpx::resource::numa_domain

Header hpx/resource_partitioner/partitioner_fwd.hpp

namespace hpx

namespace resource

Typedefs

using scheduler_function = *util*::*function_nonser*<*std*::unique_ptr<*hpx*::*threads*::*thread_pool_base*> (*hpx*::*thread*
hpx::*thread*

Enums

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.

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.

Header `hpx/runtime_configuration/agas_service_mode.hpp`

namespace hpx

namespace agas

Enums

enum service_mode
Values:

service_mode_invalid = -1

service_mode_bootstrap = 0

service_mode_hosted = 1

Header `hpx/runtime_configuration/component_registry_base.hpp`

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 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, 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 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).

Header `hpx/runtime_configuration/ini.hpp`

Defines

HPX_SECTION_VERSION

namespace hpx

namespace util

class section

#include <ini.hpp> Subclassed by *hpx::util::runtime_configuration*

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 **const** &*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* &*second*)

void **dump** (int *ind* = 0, *std*::ostream &*strm* = *std*::cout) **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** &*dflt*) **const**

template<typename **T**>
std::string **get_entry**(*std*::string **const** &*key*, *T dflt*) **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* **r*, 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*) **const**

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 *in*) **const**

void **expand** (*std*::unique_lock<*mutex_type*> &*l*, *std*::string&, *std*::string::size_type) **const**

void **expand_bracket** (*std*::unique_lock<*mutex_type*> &*l*, *std*::string&, *std*::string::size_type) **const**

void **expand_brace** (*std*::unique_lock<*mutex_type*> &*l*, *std*::string&, *std*::string::size_type) **const**

std::string **expand_only** (*std*::unique_lock<*mutex_type*> &*l*, *std*::string *in*, *std*::string **const** &*expand_this*) **const**

void **expand_only** (*std*::unique_lock<*mutex_type*> &*l*, *std*::string&, *std*::string::size_type, *std*::string **const** &*expand_this*) **const**

void **expand_bracket_only** (*std*::unique_lock<*mutex_type*> &*l*, *std*::string&, *std*::string::size_type, *std*::string **const** &*expand_this*) **const**

void **expand_brace_only** (*std*::unique_lock<*mutex_type*> &*l*, *std*::string&, *std*::string::size_type, *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

Header hpx/runtime_configuration/init_ini_data.hpp

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*::*sec* &*ini*, *std*::strin *name*, *hpx*::*uti* *get_fact* *error_cod* &*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> &*basenames*, *std*::map<*std*::string, *hpx*::*util*::*plugin*::dll> &*modules*, *std*::vector<*std*::shared_ptr<*c* &*component_registries*)

Header hpx/runtime_configuration/plugin_registry_base.hpp

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*&)

Header hpx/runtime_configuration/runtime_configuration.hpp

namespace hpx

namespace util

class runtime_configuration : **public** *hpx*::*util*::*section*
#include <runtime_configuration.hpp>

Public Functions

runtime_configuration(char **const** *argv0, runtime_mode mode)

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<componen
&compo-
nent_registries)

void **load_components_static**(std::vector<components::static_factory_load_data_type> **const** &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 **post_initialize_ini**(*std*::string &*hpx_ini_file*, *std*::vector<*std*::string> **const** &*cmdline_ini_defs*)

void **pre_initialize_logging_ini**()

void **reconfigure**()

void **load_component_paths**(*std*::vector<*std*::shared_ptr<*plugins*::*plugin_registry_base*>> &*plugin_registries*, *std*::vector<*std*::shared_ptr<*components*::*component_registry_* &*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 **load_component_path**(*std*::vector<*std*::shared_ptr<*plugins*::*plugin_registry_base*>> &*plugin_registries*, *std*::vector<*std*::shared_ptr<*components*::*component_registry_b* &*component_registries*, *std*::string **const** &*path*, *std*::set<*std*::string> &*component_paths*, *std*::map<*std*::string, *filesystem*::path> &*basenames*)

Private Members

std::string **hpx_ini_file**

std::vector<*std*::string> **cmdline_ini_defs**

std::uint32_t **num_localities**

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_**

Header `hpx/runtime_configuration/runtime_configuration_fwd.hpp`

Header `hpx/runtime_configuration/runtime_mode.hpp`

namespace hpx

Enums

enum runtime_mode

A HPX runtime can be executed in two different modes: console mode and worker mode.

Values:

invalid = -1

console = 0

The runtime is the console locality.

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

Header `hpx/runtime_configuration/static_factory_data.hpp`

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

void **init_registry_module**(*static_factory_load_data_type* **const**&)

void **init_registry_factory**(*static_factory_load_data_type* **const**&)

void **init_registry_commandline**(*static_factory_load_data_type* **const**&)

void **init_registry_startup_shutdown**(*static_factory_load_data_type* **const**&)

struct static_factory_load_data_type
#include <static_factory_data.hpp>

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.

Header `hpx/runtime_local/config_entry.hpp`

namespace hpx

Functions

std::string **get_config_entry** (*std*::string **const** &*key*, *std*::string **const** &*dflt*)
Retrieve the string value of a configuration entry given by `key`.

std::string **get_config_entry** (*std*::string **const** &*key*, *std*::size_t *dflt*)
Retrieve the integer value of a configuration entry given by `key`.

void **set_config_entry** (*std*::string **const** &*key*, *std*::string **const** &*value*)
Set the string value of a configuration entry given by `key`.

void **set_config_entry** (*std*::string **const** &*key*, *std*::size_t *value*)
Set the integer value of a configuration entry given by `key`.

void **set_config_entry_callback** (*std*::string **const** &*key*, *util*::*function_nonser*<void) *std*::string **const**&, *std*::string **const**&
> **const** &*callback*Set the string value of a configuration entry given by `key`.

Header `hpx/runtime_local/custom_exception_info.hpp`

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()* 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)

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()*, 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_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()*, hpx::get_error_backtrace(), *hpx::get_error_what()*, hpx::get_error_config(), hpx::get_error_state()

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Exceptions

- `std::bad_alloc`: (if one of the required allocations fails)

std::string **get_error_backtrace**(*hpx*::*exception_info* **const** &*xi*)
Return the stack backtrace from the point the exception was thrown.

The function hpx::get_error_backtrace can be used to extract the diagnostic information element representing the stack backtrace collected at the point the exception was thrown.

Return The stack back trace from the point the exception was thrown. If the exception instance does not hold this information, the function will return an empty string.

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_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)

std::size_t **get_error_os_thread**(*hpx*::*exception_info* **const** &*xi*)
Return the sequence number of the OS-thread used to execute HPX-threads from which the exception was thrown.

The function hpx::get_error_os_thread can be used to extract the diagnostic information element representing the sequence number of the OS-thread as stored in the given exception instance.

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 hpx::diagnostic_information(), hpx::get_error_host_name(), hpx::get_error_process_id(), *hpx::get_error_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, hpx::get_error_thread_id(), hpx::get_error_thread_description(), *hpx::get_error()*, hpx::get_error_backtrace(), hpx::get_error_env(), *hpx::get_error_what()*, hpx::get_error_config(), hpx::get_error_state()

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Exceptions

- `nothing`:

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()*, 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_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_id(), hpx::get_error_backtrace(), hpx::get_error_env(), *hpx::get_error()*, hpx::get_error_state(), *hpx::get_error_what()*, hpx::get_error_config()

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

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_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, hpx::get_error_os_thread(), hpx::get_error_thread_id(), hpx::get_error_backtrace(), hpx::get_error_env(), *hpx::get_error()*, hpx::get_error_state() *hpx::get_error_what()*, hpx::get_error_thread_description()

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

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_function_name()*, *hpx::get_error_file_name()*, *hpx::get_error_line_number()*, hpx::get_error_os_thread(), hpx::get_error_thread_id(), hpx::get_error_backtrace(), hpx::get_error_env(), *hpx::get_error()*, *hpx::get_error_what()*, hpx::get_error_thread_description()

Parameters

- `xi`: The parameter `e` will be inspected for the requested diagnostic information elements which have been stored at the point where the exception was thrown. This parameter can be one of the following types: *hpx::exception_info*, *hpx::error_code*, *std::exception*, or *std::exception_ptr*.

Exceptions

- `std::bad_alloc`: (if one of the required allocations fails)

Header `hpx/runtime_local/debugging.hpp`

`namespace hpx`

`namespace util`

Functions

void **`may_attach_debugger`** (*std*::string **`const`** &*category*)
Attaches a debugger if `category` is equal to the configuration entry hpx.attach-debugger.

Header `hpx/runtime_local/get_locality_id.hpp`

`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.

Header `hpx/runtime_local/get_num_localities.hpp`

`namespace hpx`

Functions

std::uint32_t **get_initial_num_localities**()

Return the number of localities which were registered at startup for the running application.

The function *get_initial_num_localities* returns the number of localities which were connected to the console at application startup.

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

lcos::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.

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.

Header `hpx/runtime_local/get_os_thread_count.hpp`

`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.

Header `hpx/runtime_local/get_thread_name.hpp`

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>".

Header `hpx/runtime_local/get_worker_thread_num.hpp`

Header `hpx/runtime_local/interval_timer.hpp`

namespace hpx

namespace util

class interval_timer
#include <interval_timer.hpp>

Public Functions

HPX_NON_COPYABLE(*interval_timer*)

interval_timer()

interval_timer(*util*::*function_nonser*<bool>
> **const** &*f std*::int64_t *microsecs*, *std*::string **const** &*description* = "", bool *pre_shutdown* = false

interval_timer(*util*::*function_nonser*<bool> const &*f util*::*function_nonser*<void> **const** &*on_termstd*::int64_t *microsecs*, *std*::string **const** &*description* = "", bool *pre_shutdown* = false)

interval_timer(*util*::*function_nonser*<bool> **const** &*f util*::*steady_duration* **const** &*rel_time*, char **const** **description* = "", bool *pre_shutdown* = false)

interval_timer(*util*::*function_nonser*<bool> **const** &*f util*::*function_nonser*<void> **const** &*on_termutil*::*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**(*util*::*steady_duration* **const** &*new_interval*)

Private Members

std::shared_ptr<detail::interval_timer> **timer_**

Header hpx/runtime_local/os_thread_type.hpp

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.

Header hpx/runtime_local/pool_executor.hpp

Header hpx/runtime_local/report_error.hpp

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.

Header hpx/runtime_local/run_as_hpx_thread.hpp

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*)

Header hpx/runtime_local/run_as_os_thread.hpp

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*)

Header hpx/runtime_local/runtime_handlers.hpp

Header hpx/runtime_local/runtime_local.hpp

namespace hpx

class runtime
#include <runtime_local.hpp>

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(*util*::*runtime_configuration* &*rtcfg*, bool *initialize* = true)
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.

util::*runtime_configuration* &**get_config**()
access configuration information

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 bool **is_networking_enabled**()

Return true if networking is enabled.

virtual *hpx*::*threads*::*threadmanager* &**get_thread_manager**()

Allow access to the thread manager instance used by the HPX runtime.

virtual *std*::string **here**() **const**

Returns a string of the locality endpoints (usable in debug output)

virtual bool **report_error**(*std*::size_t *num_thread*, *std*::exception_ptr **const** &*e*, bool *terminate_all* = true)

Report a non-recoverable error to the runtime system.

Parameters

- `num_thread`: [in] The number of the operating system thread the error has been detected in.
- `e`: [in] This is an instance encapsulating an exception which lead to this function call.

virtual bool **report_error**(*std*::exception_ptr **const** &*e*, bool *terminate_all* = true)

Report a non-recoverable error to the runtime system.

Note This function will retrieve the number of the current shepherd thread and forward to the report_error function above.

Parameters

- `e`: [in] This is an instance encapsulating an exception which lead to this function call.

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.

'main', 'io', 'timer', 'parcel', 'worker'

Note The function will compose a thread name of the form '<name>-thread#<num>' which is used to register the thread. It is the user's responsibility to ensure that each (composed) thread name is unique. HPX internally uses the following names for the threads it creates, do not reuse those:

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.

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.

virtual *runtime_local*::*os_thread_data* **get_os_thread_data** (*std*::string **const** &*label*) **const**
Access data for a given OS thread that was previously registered by *register_thread*.

virtual bool **enumerate_os_threads** (*util*::*function_nonser*<bool) *runtime_local*::*os_thread_data* **const**& > **const** &*f* **const**Enumerate all OS threads that have registered with the runtime.

notification_policy_type::on_startstop_type **on_start_func** () **const**

notification_policy_type::on_startstop_type **on_stop_func** () **const**

notification_policy_type::on_error_type **on_error_func** () **const**

notification_policy_type::on_startstop_type **on_start_func** (*notification_policy_type*::on_startstop_type&&)

notification_policy_type::on_startstop_type **on_stop_func** (*notification_policy_type*::on_startstop_type&&)

notification_policy_type::on_error_type **on_error_func** (*notification_policy_type*::on_error_type&&)

virtual *std*::uint32_t **get_locality_id** (*error_code* &*ec*) **const**

virtual *std*::size_t **get_num_worker_threads** () **const**

virtual *std*::uint32_t **get_num_localities** (*hpx*::*launch*::sync_policy, *error_code* &*ec*) **const**

virtual *std*::uint32_t **get_initial_num_localities**() **const**

virtual *lcos*::future<*std*::uint32_t> **get_num_localities**() **const**

Public Static Functions

static *std*::uint64_t **get_system_uptime**()
Return the system uptime measure on the thread executing this call.

Protected Types

using on_exit_type = *std*::vector<*util*::*function_nonser*<void()>>

Protected Functions

runtime(*util*::*runtime_configuration* &*rtcfg*, *notification_policy_type* &&*notifier*, *notification_policy_type* &&*main_pool_notifier*, bool *initialize*)

void **init**()
Common initialization for different constructors.

void **init_tss**()

void **deinit_tss**()

threads::thread_result_type **run_helper**(*util*::*function_nonser*<*runtime*::*hpx_main_function_type*> **const** &*func*, int &*result*, bool *call_startup_functions*)

void **wait_helper**(*std*::mutex &*mtx*, *std*::condition_variable &*cond*, bool &*running*)

Protected Attributes

on_exit_type **on_exit_functions_**

std::mutex **mtx_**

util::*runtime_configuration* **ini_**

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_**

std::unique_ptr<*hpx*::*threads*::*threadmanager*> **thread_manager_**

Protected Static Attributes

std::atomic<int> **instance_number_counter_**

Private Functions

void **stop_helper**(bool *blocking*, *std*::condition_variable &*cond*, *std*::mutex &*mtx*)
Helper function to stop the runtime.

Parameters
- `blocking`: [in] This allows to control whether this call blocks until the runtime system has been fully stopped. If this parameter is *false* then this call will initiate the stop action but will return immediately. Use a second call to stop with this parameter set to *true* to wait for all internal work to be completed.

void **deinit_tss_helper**(char **const** **context*, *std*::size_t *num*)

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*)

void **init_tss_helper**(char **const** **context*, *runtime_local*::*os_thread_type type*, *std*::size_t *local_thread_num*, *std*::size_t *global_thread_num*, char **const** **pool_name*, char **const** **postfix*, bool *service_thread*)

void **notify_finalize**()

void **wait_finalize**()

runtime ***This**()

void **call_startup_functions**(bool *pre_startup*)

Private Members

std::list<*startup_function_type*> **pre_startup_functions_**

std::list<*startup_function_type*> **startup_functions_**

std::list<*shutdown_function_type*> **pre_shutdown_functions_**

std::list<*shutdown_function_type*> **shutdown_functions_**

bool **stop_called_**

bool **stop_done_**

std::condition_variable **wait_condition_**

namespace threads

Functions

char **const** ***get_stack_size_name**(*std*::ptrdiff_t *size*)

Returns the stack size name.

Get the readable string representing the given stack size constant.

Parameters

- `size`: this represents the stack size

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*)

Header `hpx/runtime_local/runtime_local_fwd.hpp`

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** &*f*Enumerate all OS threads that have registered with the runtime.

std::size_t **get_runtime_instance_number**()

Return the runtime instance number associated with the runtime instance the current thread is running in.

bool **register_on_exit**(*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.

Header hpx/runtime_local/service_executors.hpp

namespace hpx

namespace parallel

namespace execution

Enums

enum service_executor_type
Values:

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*
#include <service_executors.hpp>

Public Functions

io_pool_executor()

struct main_pool_executor : **public** *service_executor*
#include <service_executors.hpp>

Public Functions

main_pool_executor()

struct parcel_pool_executor : **public** *service_executor*
#include <service_executors.hpp>

Public Functions

parcel_pool_executor(char **const** **name_suffix* = "-tcp")

struct service_executor : **public** *service_executor*
#include <service_executors.hpp>

Public Functions

service_executor(*service_executor_type t*, char **const** **name_suffix* = "")

struct timer_pool_executor : **public** *service_executor*
#include <service_executors.hpp>

Public Functions

timer_pool_executor()

Header hpx/runtime_local/shutdown_function.hpp

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 *hpx::finalize()* but guaranteed before any shutdown function is executed (system-wide)

Any of the functions registered with *register_pre_shutdown_function* are guaranteed to be executed by an HPX thread during the execution of *hpx::finalize()* before any of the registered shutdown functions are executed (see: hpx::register_shutdown_function()).

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.

See hpx::register_shutdown_function()

Parameters

- `f`: [in] The function to be registered to run by an HPX thread as a pre-shutdown function.

void **register_shutdown_function**(*shutdown_function_type f*)
Add a function to be executed by a HPX thread during *hpx::finalize()* but guaranteed after any pre-shutdown function is executed (system-wide)

Any of the functions registered with *register_shutdown_function* are guaranteed to be executed by an HPX thread during the execution of *hpx::finalize()* after any of the registered pre-shutdown functions are executed (see: hpx::register_pre_shutdown_function()).

Note If this function is called while the shutdown functions are being executed, or after that point, it will raise a invalid_status exception.

See hpx::register_pre_shutdown_function()

Parameters

- `f`: [in] The function to be registered to run by an HPX thread as a shutdown function.

Header hpx/runtime_local/startup_function.hpp

namespace hpx

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()

Header `hpx/runtime_local/state.hpp`

namespace hpx

namespace threads

Functions

bool **threadmanager_is**(*state st*)

bool **threadmanager_is_at_least**(*state st*)

Header `hpx/runtime_local/thread_hooks.hpp`

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_no*
&&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.

threads::policies::callback_notifier::on_startstop_type **`register_thread_on_stop_func`**(*threads::policies::callback_not*
&&f)

Set the currently installed stop handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered stop function chains into the previous one (see *get_thread_on_stop_func*).

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.

threads::policies::callback_notifier::on_error_type **`register_thread_on_error_func`**(*threads::policies::callback_notifi*
&&f)

Set the currently installed error handler function. This is a function that will be called by HPX for each newly created thread that is made known to the runtime. HPX stores exactly one such function reference, thus the caller needs to make sure any newly registered error function chains into the previous one (see *get_thread_on_error_func*).

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.

Header `hpx/runtime_local/thread_mapper.hpp`

namespace hpx

namespace util

class thread_mapper

#include <thread_mapper.hpp>

Public Types

using callback_type = detail::thread_mapper_callback_type

Public Functions

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

constexpr *std*::uint32_t **invalid_index** = *std*::uint32_t(-1)

constexpr *std*::uint64_t **invalid_tid** = *std*::uint64_t(-1)

Private Types

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

mutex_type **mtx_**

thread_map_type **thread_map_**

label_map_type **label_map_**

Header `hpx/runtime_local/thread_pool_helpers.hpp`

namespace hpx

namespace resource

Functions

std::size_t **get_num_thread_pools**()
Return the number of thread pools currently managed by the *resource_partitioner*

std::size_t **get_num_threads**()
Return the number of threads in all thread pools currently managed by the *resource_partitioner*

std::size_t **get_num_threads**(*std*::string **const** &*pool_name*)
Return the number of threads in the given thread pool currently managed by the *resource_partitioner*

std::size_t **get_num_threads**(*std*::size_t *pool_index*)
Return the number of threads in the given thread pool currently managed by the *resource_partitioner*

std::size_t **get_pool_index**(*std*::string **const** &*pool_name*)
Return the internal index of the pool given its name.

std::string **const** &**get_pool_name**(*std*::size_t *pool_index*)
Return the name of the pool given its internal index.

threads::*thread_pool_base* &**get_thread_pool**(*std*::string **const** &*pool_name*)
Return the name of the pool given its name.

threads::*thread_pool_base* &**get_thread_pool**(*std*::size_t *pool_index*)
Return the thread pool given its internal index.

bool **pool_exists**(*std*::string **const** &*pool_name*)
Return true if the pool with the given name exists.

bool **pool_exists**(*std*::size_t *pool_index*)
Return true if the pool with the given index exists.

namespace threads

Functions

std::int64_t **get_thread_count** (*thread_state_enum state* = *unknown*)
The function *get_thread_count* returns the number of currently known threads.

Note If state == unknown this function will not only return the number of currently existing threads, but will add the number of registered task descriptions (which have not been converted into threads yet).

Parameters
- `state`: [in] This specifies the thread-state for which the number of threads should be retrieved.

std::int64_t **get_thread_count** (*thread_priority priority*, *thread_state_enum state* = *unknown*)
The function *get_thread_count* returns the number of currently known threads.

Note If state == unknown this function will not only return the number of currently existing threads, but will add the number of registered task descriptions (which have not been converted into threads yet).

Parameters
- `priority`: [in] This specifies the thread-priority for which the number of threads should be retrieved.
- `state`: [in] This specifies the thread-state for which the number of threads should be retrieved.

std::int64_t **get_idle_core_count** ()
The function *get_idle_core_count* returns the number of currently idling threads (cores).

mask_type **get_idle_core_mask** ()
The function *get_idle_core_mask* returns a bit-mask representing the currently idling threads (cores).

bool **enumerate_threads** (*util*::*function_nonser*<bool) *thread_id_type*
> **const** &*f*, *thread_state_enum state* = *unknown*The function *enumerate_threads* will invoke the given function *f* for each thread with a matching thread state.

Parameters
- `f`: [in] The function which should be called for each matching thread. Returning 'false' from this function will stop the enumeration process.
- `state`: [in] This specifies the thread-state for which the threads should be enumerated.

Header `hpx/runtime_local/thread_stacktrace.hpp`

namespace hpx

namespace util

namespace debug

Functions

std::vector<*hpx*::*threads*::*thread_id_type*> **get_task_ids** (*hpx*::*threads*::*thread_state_enum state* = *hpx*::*threads*::*suspended*)

std::vector<*hpx*::*threads*::*thread_data**> **get_task_data** (*hpx*::*threads*::*thread_state_enum state* = *hpx*::*threads*::*suspended*)

std::string **suspended_task_backtraces** ()

Header `hpx/util/thread_aware_timer.hpp`

schedulers

The contents of this module can be included with the header `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 `hpx/modules/schedulers.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.

Header `hpx/modules/schedulers.hpp`

Header `hpx/schedulers/deadlock_detection.hpp`

Header `hpx/schedulers/local_priority_queue_scheduler.hpp`

namespace hpx

namespace threads

namespace policies

Typedefs

using default_local_priority_queue_scheduler_terminated_queue = *lockfree_fifo*

template<typename **Mutex** = *std*::mutex, typename **PendingQueuing** = *lockfree_fifo*, typename **StagedQueuing**
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_typ**

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_type* **id*, *error_code* &*ec*)

bool **get_next_thread**(*std*::size_t *num_thread*, bool *running*, *threads*::*thread_data* *&*thrd*, bool *enable_stealing*)
Return the next thread to be executed, return false if none is available

void **schedule_thread**(*threads*::*thread_data* **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_data* **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_state_enum state* = *unknown*, *thread_priority priority* = *thread_priority_default*, *std*::size_t *num_thread* = *std*::size_t(-1), bool *reset* = 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_state_enum 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_**

std::size_t **const num_queues_**

std::size_t **const 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
#include <local_priority_queue_scheduler.hpp>

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

template<>
std::size_t **num_queues_**

template<>
std::size_t **num_high_priority_queues_**

template<>
thread_queue_init_parameters **thread_queue_init_**

template<>
detail::affinity_data **const** &**affinity_data_**

template<>
char **const** ***description_***

Header `hpx/schedulers/local_queue_scheduler.hpp`

Header `hpx/schedulers/lockfree_queue_backends.hpp`

namespace hpx

namespace threads

namespace policies

struct concurrentqueue_fifo
#include <lockfree_queue_backends.hpp>

template<typename **T**>
struct apply
#include <lockfree_queue_backends.hpp>

Public Types

template<>
using type = moodycamel_fifo_backend<T>

struct lockfree_fifo
#include <lockfree_queue_backends.hpp>

template<typename **T**>
struct apply
#include <lockfree_queue_backends.hpp>

Public Types

template<>
using type = lockfree_fifo_backend<T>

template<typename **T**>
struct lockfree_fifo_backend
#include <lockfree_queue_backends.hpp>

Public Types

template<>
using container_type = *boost*::*lockfree*::queue<T>

template<>
using value_type = T

template<>
using reference = T&

template<>
using const_reference = T **const**&

template<>
using size_type = *std*::uint64_t

Public Functions

lockfree_fifo_backend (size_type *initial_size* = 0, size_type *num_thread* = size_type(-1))

bool **push** (const_reference *val*, bool = false)

bool **pop** (reference *val*, bool *steal* = true)

bool **empty** ()

Private Members

container_type **queue_**

template<typename **T**>
struct moodycamel_fifo_backend
#include <lockfree_queue_backends.hpp>

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 rval_reference = T&&

template<>
using size_type = *std*::uint64_t

Public Functions

moodycamel_fifo_backend (size_type *initial_size* = 0, size_type *num_thread* = size_type(-1))

bool **push** (rval_reference *val*, bool = false)

bool **push** (const_reference *val*, bool = false)

bool **pop** (reference *val*, bool *steal* = true)

bool **empty** ()

Private Members

container_type **queue_**

Header hpx/schedulers/maintain_queue_wait_times.hpp

Header hpx/schedulers/queue_helpers.hpp

Header hpx/schedulers/queue_holder_numa.hpp

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
#include <queue_holder_numa.hpp>

Public Types

template<>
using ThreadQueue = queue_holder_thread<QueueType>

template<>
using mutex_type = **typename** QueueType::mutex_type

Public Functions

queue_holder_numa()

~queue_holder_numa()

void **init**(*std*::size_t *domain*, *std*::size_t *queues*)

std::size_t **size**() **const**

ThreadQueue ***thread_queue**(*std*::size_t *id*) **const**

bool **get_next_thread_HP**(*std*::size_t *qidx*, *threads*::*thread_data* *&*thrd*, bool *stealing*, bool *core_stealing*)

bool **get_next_thread**(*std*::size_t *qidx*, *threads*::*thread_data* *&*thrd*, bool *stealing*, bool *core_stealing*)

bool **add_new_HP**(ThreadQueue **receiver*, *std*::size_t *qidx*, *std*::size_t &*added*, bool *stealing*, bool *allow_stealing*)

bool **add_new**(ThreadQueue **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_state_enum 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_state_enum state* **const**

void **increment_num_pending_misses**(*std*::size_t *num* = 1)

void **increment_num_pending_accesses**(*std*::size_t *num* = 1)

void **increment_num_stolen_from_pending**(*std*::size_t *num* = 1)

void **increment_num_stolen_from_staged**(*std*::size_t *num* = 1)

void **increment_num_stolen_to_pending**(*std*::size_t *num* = 1)

void **increment_num_stolen_to_staged**(*std*::size_t *num* = 1)

bool **dump_suspended_threads**(*std*::size_t *num_thread*, *std*::int64_t &*idle_loop_count*, bool *running*)

void **debug_info**()

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 Members

std::size_t **num_queues_**

std::size_t **domain_**

std::vector<ThreadQueue*> **queues_**

Header hpx/schedulers/queue_holder_thread.hpp

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
#include <queue_holder_thread.hpp>

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<*th*

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_data* **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_type* **tid*, *std*::size_t *thread_num*, *error_code* &*ec*)

void **create_thread_object**(*threads*::*thread_id_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_data* *&*thrd*, bool *stealing*, bool *check_new*) bool **get_next_thread**(*threads*::*thread_data* *&*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_state_enum 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) *thread_id_type*
> **const** &*f*, *thread_state_enum 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
#include <queue_holder_thread.hpp>

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
#include <queue_holder_thread.hpp>

Public Functions

template<>
queue_mc_print(**const** QueueType ***const** *q*)

Public Members

template<>
const QueueType ***const q_**

Friends

std::ostream &**operator<<** (*std*::ostream &*os*, **const** queue_mc_print &*d*)

Header `hpx/schedulers/shared_priority_queue_scheduler.hpp`

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

#include <shared_priority_queue_scheduler.hpp>

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 **Termina**
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, TerminatedQu

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 *thread_num*, bool *delete_all*)

void **create_thread** (*thread_init_data* &*data*, *thread_id_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_data* *&*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 *thread_num*, bool *running*, *std*::int64_t &*idle_loop_count*, bool, *std*::size_t &*added*)
Return the next thread to be executed, return false if none available.

void **schedule_thread**(*threads*::*thread_data* **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_data* **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_state_enum* *state* = *unknown*, *thread_priority* *priority* = *thread_priority_default*, *std*::size_t *thread_num* = *std*::size_t(-1), bool *reset* = 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_state_enum* *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** &*e*)

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::array<*std*::size_t, HPX_HAVE_MAX_CPU_COUNT> **d_lookup_**

std::array<*std*::size_t, HPX_HAVE_MAX_CPU_COUNT> **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**

volatile bool **initialized_**

volatile bool **debug_init_**

volatile *std*::size_t **thread_init_counter_**

std::size_t **pool_index_**

struct init_parameter
#include <shared_priority_queue_scheduler.hpp>

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")

template<>
init_parameter (*std*::size_t *num_worker_threads*, **const** *core_ratios* &*cores_per_queue*, detail::affinity_data **const** &*affinity_data*, char **const** **description*)

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_**

Header hpx/schedulers/static_priority_queue_scheduler.hpp

Header hpx/schedulers/static_queue_scheduler.hpp

Header hpx/schedulers/thread_queue.hpp

namespace hpx

namespace threads

namespace policies

template<typename **Mutex**, typename **PendingQueuing**, typename **StagedQueuing**, typename **TerminatedQ**
class thread_queue
#include <thread_queue.hpp>

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**

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 *num* = 1)

constexpr void **increment_num_pending_accesses** (*std*::size_t *num* = 1)

constexpr void **increment_num_stolen_from_pending** (*std*::size_t *num* = 1)

constexpr void **increment_num_stolen_from_staged** (*std*::size_t *num* = 1)

constexpr void **increment_num_stolen_to_pending** (*std*::size_t *num* = 1)

constexpr void **increment_num_stolen_to_staged** (*std*::size_t *num* = 1)

void **create_thread** (*thread_init_data &data*, *thread_id_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_data* *&*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_data* **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_state_enum 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_state_enum state* = *unknown* **const**

bool **wait_or_add_new** (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).

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 *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 void **deallocate** (*threads*::*thread_data* **p*)

Protected Functions

template<typename **Lock**>
void **create_thread_object** (*threads*::*thread_id_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*<**typename** 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<*th*

template<>
using thread_heap_type = *std*::list<*thread_id_type*, *util*::*internal_allocator*<*thread_id_type*>>

template<>
using thread_description = *thread_data*

template<>
using work_items_type = **typename** PendingQueuing::template apply<*thread_description**>::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_**

std::atomic<*std*::int64_t> **terminated_items_count_**

task_items_type **new_tasks_**

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*::atomic<*std*::int64_t>> **new_tasks_count_**

util::*cache_line_data*<*std*::atomic<*std*::int64_t>> **work_items_count_**

struct task_description

Public Members

template<>
thread_init_data **data**

Header `hpx/schedulers/thread_queue_mc.hpp`

Defines

THREAD_QUEUE_MC_DEBUG

namespace hpx

Functions

static hpx::debug::enable_print<THREAD_QUEUE_MC_DEBUG> hpx::tqmc_deb("_TQ_MC_")

namespace threads

namespace policies

template<typename **Mutex**, typename **PendingQueuing**, typename **StagedQueuing**, typename **TerminatedQ**
class thread_queue_mc
#include <thread_queue_mc.hpp>

Public Types

typedef Mutex **mutex_type**

template<>
using thread_queue_type = thread_queue_mc<Mutex, PendingQueuing, StagedQueuing, TerminatedQueu

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_description = *thread_data*

typedef PendingQueuing::template apply<thread_description*>::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*)

thread_queue_mc(**const** *thread_queue_init_parameters* &*parameters*, *std*::size_t *queue_num* = *std*::size_t(-1))

void **set_holder**(queue_holder_thread<thread_queue_type> **holder*)

~thread_queue_mc()

std::int64_t **get_queue_length**() **const**

std::int64_t **get_queue_length_pending**() **const**

std::int64_t **get_queue_length_staged**(*std*::memory_order *order* = *std*::memory_order_seq_cst) **const**

std::int64_t **get_thread_count**() **const**

void **create_thread**(*thread_init_data* &*data*, *thread_id_type* **id*, *error_code* &*ec*)

bool **get_next_thread**(*threads*::*thread_data* *&*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_data* **thrd*, bool *other_end*)
Schedule the passed thread (put it on the ready work queue)

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 Members

thread_queue_init_parameters **parameters_**

int **const queue_index_**

queue_holder_thread<thread_queue_type> ***holder_**

task_items_type **new_task_items_**

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_**

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.

Header `hpx/parallel/segmented_algorithm.hpp`

Header `hpx/parallel/segmented_algorithms/adjacent_difference.hpp`

Header `hpx/parallel/segmented_algorithms/adjacent_find.hpp`

Header `hpx/parallel/segmented_algorithms/all_any_none.hpp`

Header `hpx/parallel/segmented_algorithms/count.hpp`

Header `hpx/parallel/segmented_algorithms/exclusive_scan.hpp`

Header `hpx/parallel/segmented_algorithms/fill.hpp`

Header `hpx/parallel/segmented_algorithms/find.hpp`

Header `hpx/parallel/segmented_algorithms/for_each.hpp`

Header `hpx/parallel/segmented_algorithms/generate.hpp`

Header `hpx/parallel/segmented_algorithms/inclusive_scan.hpp`

Header `hpx/parallel/segmented_algorithms/minmax.hpp`

Header `hpx/parallel/segmented_algorithms/reduce.hpp`

Header `hpx/parallel/segmented_algorithms/traits/zip_iterator.hpp`

Header `hpx/parallel/segmented_algorithms/transform.hpp`

Header `hpx/parallel/segmented_algorithms/transform_exclusive_scan.hpp`

Header `hpx/parallel/segmented_algorithms/transform_inclusive_scan.hpp`

Header `hpx/parallel/segmented_algorithms/transform_reduce.hpp`

serialization

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.

Header hpx/serialization.hpp

Header hpx/serialization/access.hpp

template<typename **T**>
struct serialize_non_intrusive<*T*, **typename** *std*::enable_if<has_serialize_adl<*T*>::value>::type>
#include <access.hpp>

Public Static Functions

template<typename **Archive**>
static void **call** (*Archive* &*ar*, T &*t*, unsigned)

namespace hpx

namespace serialization

class access
#include <access.hpp>

Public Static Functions

template<class **Archive**, class **T**>
static void **serialize** (*Archive* &*ar*, *T* &*t*, unsigned)

template<typename **Archive**, typename **T**>
static void **save_base_object** (*Archive* &*ar*, *T* **const** &*t*, unsigned)

template<typename **Archive**, typename **T**>
static void **load_base_object** (*Archive* &*ar*, *T* &*t*, unsigned)

template<typename **T**>
static *std*::string **get_name** (*T* **const** **t*)

template<class **T**>
class has_serialize

Public Static Attributes

constexpr bool **value** = decltype(test<T>(0))::value

Private Static Functions

template<class **T1**>
static *std*::false_type **test** (...)

template<class **T1**, class = decltype(*std*::declval<**typename** *std*::remove_const<*T1*>::type&>().serialize(*std*::decl
static *std*::true_type **test** (int)

template<class **T**>
class serialize_dispatcher

Public Types

template<>
using type = **typename** *std*::conditional::type

struct empty

Public Static Functions

template<class **Archive**>
static void **call** (Archive&, T&, unsigned)

struct intrusive_polymorphic

Public Static Functions

template<>
static void **call** (*hpx*::*serialization*::*input_archive* &*ar*, T &*t*, unsigned)

template<>
static void **call** (*hpx*::*serialization*::*output_archive* &*ar*, T **const** &*t*, unsigned)

struct intrusive_usual

Public Static Functions

template<class **Archive**>
static void **call** (Archive &*ar*, T &*t*, unsigned)

struct non_intrusive

Public Static Functions

template<class **Archive**>
static void **call** (Archive &*ar*, T &*t*, unsigned)

template<typename **T**>
class has_serialize_adl
#include <access.hpp>

Public Static Attributes

constexpr bool **value** = decltype(test<T>(0))::value

Private Static Functions

template<typename **T1**>
static *std*::false_type **test** (...)

template<typename **T1**, typename = decltype(*serialize*(*std*::declval<*hpx*::*serialization*::*output_archive*&>(), *std*::declva
static *std*::true_type **test** (int)

template<typename **T**>
struct serialize_non_intrusive<*T*, **typename** *std*::enable_if<has_serialize_adl<*T*>::value>::type>
#include <access.hpp>

Public Static Functions

template<typename **Archive**>
static void **call** (*Archive* &*ar*, T &*t*, unsigned)

Header `hpx/serialization/array.hpp`

namespace hpx

namespace serialization

Functions

template<class **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<class **T**>
class array
#include <array.hpp>

Public Types

template<>
using value_type = T

Public Functions

array(value_type **t*, *std*::size_t *s*)

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**

Header hpx/serialization/base_object.hpp

template<typename **Derived**, typename **Base**>
struct base_object_type<*Derived*, *Base*, *std*::true_type>
#include <base_object.hpp>

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*)

template<typename **D**, typename **B**>
output_archive &**operator&**(*output_archive* &*ar*, base_object_type<*D*, *B*> *t*)

template<typename **D**, typename **B**>
input_archive &**operator&**(*input_archive* &*ar*, base_object_type<*D*, *B*> *t*)

template<typename **Derived**, typename **Base**, typename **Enable** = **typename** *hpx*::*traits*::is_intrusive_polymorphic<*I*
struct base_object_type
#include <base_object.hpp>

Public Functions

base_object_type (Derived &*d*)

template<typename **Archive**>
void **serialize** (*Archive* &*ar*, unsigned)

Public Members

Derived &**d_**

template<typename **Derived**, typename **Base**>
struct base_object_type<*Derived*, *Base*, *std*::true_type>
#include <base_object.hpp>

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_**

Header `hpx/serialization/basic_archive.hpp`

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
#include <basic_archive.hpp>

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_**

Header `hpx/serialization/binary_filter.hpp`

namespace hpx

namespace serialization

struct binary_filter

#include <binary_filter.hpp>

Public Functions

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 ()

Header hpx/serialization/bitset.hpp

namespace hpx

namespace serialization

Functions

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)

Header hpx/serialization/boost_variant.hpp

Header hpx/serialization/brace_initializable.hpp

Header hpx/serialization/brace_initializable_fwd.hpp

Header hpx/serialization/complex.hpp

namespace hpx

namespace serialization

Functions

template<typename **T**>
void **serialize**(*input_archive* &*ar*, *std*::complex<*T*> &*c*, unsigned)

template<typename **T**>
void **serialize**(*output_archive* &*ar*, *std*::complex<*T*> **const** &*c*, unsigned)

Header hpx/serialization/container.hpp

namespace hpx

namespace serialization

struct erased_input_container
#include <container.hpp> 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

#include <container.hpp> 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

Header hpx/serialization/datapar.hpp

Header hpx/serialization/deque.hpp

namespace hpx

namespace serialization

Functions

template<typename **T**, typename **Allocator**>
void **serialize**(*input_archive* &*ar*, *std*::deque<*T*, *Allocator*> &*d*, unsigned)

template<typename **T**, typename **Allocator**>
void **serialize**(*output_archive* &*ar*, *std*::deque<*T*, *Allocator*> **const** &*d*, unsigned)

Header `hpx/serialization/dynamic_bitset.hpp`

Header `hpx/serialization/exception_ptr.hpp`

namespace hpx

namespace serialization

Functions

template<typename **Archive**>
void **save** (*Archive* &*ar*, *std*::exception_ptr **const** &*e*, unsigned int)

template<typename **Archive**>
void **load** (*Archive* &*ar*, *std*::exception_ptr &*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

Header hpx/serialization/input_archive.hpp

namespace hpx

namespace serialization

struct input_archive : **public** *hpx*::*serialization*::basic_archive<*input_archive*>
#include <input_archive.hpp>

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<!*std*::is_integral<*T*>::value && !*std*::is_enum<*T*>::value>::type **load**(*T* &*t*)

template<typename **T**>
std::enable_if<*std*::is_integral<*T*>::value || *std*::is_enum<*T*>::value>::type **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

Header `hpx/serialization/input_container.hpp`

namespace hpx

namespace serialization

template<typename **Container**>
struct input_container : **public** *hpx*::*serialization*::*erased_input_container*
#include <input_container.hpp>

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**

Header hpx/serialization/intrusive_ptr.hpp

Header hpx/serialization/list.hpp

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)

Header hpx/serialization/map.hpp

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)

Header hpx/serialization/multi_array.hpp

Header hpx/serialization/optional.hpp

namespace hpx

namespace serialization

Functions

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)

hpx::serialization::HPX_SERIALIZATION_SPLIT_FREE_TEMPLATE((template< typename T >),

Header hpx/serialization/output_archive.hpp

namespace hpx

namespace serialization

struct output_archive : **public** *hpx*::*serialization*::basic_archive<*output_archive*>
#include <output_archive.hpp>

Public Types

using base_type = basic_archive<*output_archive*>

Public Functions

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

template<typename **T**>
void **invoke_impl**(*T* **const** &*t*)

template<typename **T**>
std::enable_if<!*std*::is_integral<*T*>::value && !*std*::is_enum<*T*>::value>::type **save**(*T* **const** &*t*)

template<typename **T**>
std::enable_if<*std*::is_integral<*T*>::value || *std*::is_enum<*T*>::value>::type **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

Header `hpx/serialization/output_container.hpp`

namespace hpx

namespace serialization

template<typename **Container**, typename **Chunker**>
struct filtered_output_container : **public** *hpx*::*serialization*::output_container<*Container*, *Chunker*>
#include <output_container.hpp>

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*

#include <output_container.hpp> 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 *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_**

Header `hpx/serialization/serializable_any.hpp`

template<typename **IArch**, typename **OArch**, typename **Char**>
class basic_any<*IArch*, *OArch*, *Char*, *std*::true_type>
#include <serializable_any.hpp>

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*>::ty
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_copy_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

~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*>::ty
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

namespace util

Typedefs

using any = 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**>
basic_any<*serialization*::*input_archive*, *serialization*::*output_archive*, *Char*> **make_any**(*T* &&*t*)

template<typename **IArch**, typename **OArch**, typename **Char**>
class basic_any<*IArch*, *OArch*, *Char*, *std*::true_type>
#include <serializable_any.hpp>

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*::dec
basic_any(*T* &&*x*, **typename** *std*::enable_if<*std*::is_copy_constructible<**typename** *std*::decay<*T*>::type>::value>::type* = nullptr)

~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*::dec
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::util::hpx::serialization::access

struct hash_any
#include <serializable_any.hpp>

Public Functions

template<typename **Char**>
std::size_t **operator()** (**const** basic_any<*serialization*::*input_archive*, *serialization*::*output_archive*, *Char*, *std*::true_type> &*elem*) **const**

Header `hpx/serialization/serialization_chunk.hpp`

namespace hpx

namespace serialization

Enums

enum chunk_type
Values:

chunk_type_index = 0

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
#include <serialization_chunk.hpp>

Public Members

std::size_t **index_**

void **const** *__cpos_**

void *__pos_**

struct serialization_chunk
#include <serialization_chunk.hpp>

Public Members

chunk_data **data_**

std::size_t **size_**

std::uint64_t **rkey_**

std::uint8_t **type_**

Header hpx/serialization/serialization_fwd.hpp

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*)

Header `hpx/serialization/serialize.hpp`

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*)

Header `hpx/serialization/serialize_buffer.hpp`

namespace hpx

namespace serialization

template<typename **T**, typename **Allocator** = *std*::allocator<*T*>>
class serialize_buffer
#include <serialize_buffer.hpp>

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<typename **Deallocator**>
serialize_buffer(T **data*, *std*::size_t *size*, allocator_type **const** &*alloc*, *Deallocator* **const** &*dealloc*)

template<typename **Deleter**>

serialize_buffer (T **data*, *std*::size_t *size*, *init_mode mode*, *Deleter* **const** &*deleter*, allocator_type **const** &*alloc* = allocator_type())

template<typename **Deleter**>
serialize_buffer (T **const** **data*, *std*::size_t *size*, *init_mode mode*, *Deleter* **const** &*deleter*, allocator_type **const** &*alloc* = allocator_type())

template<typename **Deallocator**, typename **Deleter**>
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())

template<typename **Deleter**>
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**

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*)

Header `hpx/serialization/set.hpp`

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)

Header `hpx/serialization/shared_ptr.hpp`

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)

Header `hpx/serialization/std_tuple.hpp`

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* &*ar*, *std*::tuple<>&, unsigned int)

Header `hpx/serialization/string.hpp`

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)

Header `hpx/serialization/traits/brace_initializable_traits.hpp`

Header `hpx/serialization/traits/is_bitwise_serializable.hpp`

Defines

HPX_IS_BITWISE_SERIALIZABLE (*T*)

Header hpx/serialization/traits/needs_automatic_registration.hpp

Header hpx/serialization/traits/polymorphic_traits.hpp

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*)

Header hpx/serialization/traits/serialization_access_data.hpp

namespace hpx

namespace traits

template<typename **Container**>
struct default_serialization_access_data
#include <serialization_access_data.hpp> 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 &*cont*, *std*::size_t *count*, *std*::size_t *current*, void **const** **address*)

static bool **flush** (*serialization*::*binary_filter* **filter*, Container &*cont*, *std*::size_t *current*, *std*::size_t *size*, *std*::size_t &*written*)

static constexpr void **read** (Container **const** &*cont*, *std*::size_t *count*, *std*::size_t *current*, void **address*)

static constexpr *std*::size_t **init_data** (Container **const** &*cont*, *serialization*::*binary_filter* **filter*, *std*::size_t *current*, *std*::size_t *decompressed_size*)

static constexpr void **reset** (Container &*cont*)

template<typename **Container**>
struct serialization_access_data : **public** *hpx*::*traits*::default_serialization_access_data<*Container*>
#include <serialization_access_data.hpp> Subclassed by hpx::traits::serialization_access_data< Container const >

Public Static Functions

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 **const** **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*)

Header `hpx/serialization/tuple.hpp`

namespace hpx

namespace serialization

Functions

template<typename **Archive**, typename ...**Ts**>
void **serialize**(*Archive* &*ar*, *hpx*::*util*::tuple<*Ts*...> &*t*, unsigned int *version*)

template<typename **Archive**>
void **serialize**(*Archive* &*ar*, *hpx*::*util*::*tuple*<>&, unsigned)

template<typename **Archive**, typename ...**Ts**>
void **load_construct_data**(*Archive* &*ar*, *hpx*::*util*::tuple<*Ts*...> **t*, unsigned int *version*)

template<typename **Archive**, typename ...**Ts**>
void **save_construct_data**(*Archive* &*ar*, *hpx*::*util*::tuple<*Ts*...> **const** **t*, unsigned int *version*)

Header `hpx/serialization/unique_ptr.hpp`

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)

Header `hpx/serialization/unordered_map.hpp`

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)

template<typename **Key**, typename **Value**, typename **Hash**, typename **KeyEqual**, typename **Alloc**>
void **serialize**(*output_archive* &*ar*, **const** *std*::unordered_map<*Key*, *Value*, *Hash*, *KeyEqual*, *Alloc*> &*t*, unsigned)

Header `hpx/serialization/valarray.hpp`

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)

Header `hpx/serialization/variant.hpp`

Header `hpx/serialization/vector.hpp`

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)

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.

Header `hpx/static_reinit/reinitializable_static.hpp`

Defines

HPX_EXPORT_REINITIALIZABLE_STATIC

namespace hpx

namespace util

Variables

template<typename **T**, typename **Tag** = *T*, *std*::size_t **N** = 1>
struct HPX_EXPORT_REINITIALIZABLE_STATIC **reinitializable_static**

template<typename **T**, typename **Tag**, *std*::size_t **N**>
struct reinitializable_static
#include <reinitializable_static.hpp>

Public Types

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_**

Header `hpx/static_reinit/static_reinit.hpp`

namespace hpx

namespace util

Functions

void **reinit_register**(*util*::*function_nonser*<void> **const** &*construct*, *util*::*function_nonser*<void> **const** &*destruct*)

void **reinit_construct**()

void **reinit_destruct**()

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.

Header `hpx/statistics/histogram.hpp`

namespace boost

namespace accumulators

namespace extract

Variables

extractor<*tag*::*histogram*> **const histogram** = {}

namespace tag

struct histogram : **public** depends_on<count>, **public** histogram_num_bins, **public** histogram_min_rang

#include <histogram.hpp>

struct impl

#include <histogram.hpp>

template<typename **Sample**, typename **Weight**>
struct apply

#include <histogram.hpp>

Public Types

typedef *hpx*::*util*::detail::histogram_impl<Sample> **type**

Header hpx/statistics/max.hpp

namespace hpx

namespace util

Functions

template<typename T>constexpr T const&() hpx::util::max(T const & a, T const & b)

Header hpx/statistics/min.hpp

namespace hpx

namespace util

Functions

template<typename T>constexpr T const&() hpx::util::min(T const & a, T const & b)

Header hpx/statistics/rolling_max.hpp

namespace boost

namespace accumulators

namespace extract

Variables

extractor<*tag*::*rolling_max*> **const rolling_max** = { }

namespace tag

struct rolling_max : **public** depends_on<rolling_window>
#include <rolling_max.hpp>

struct impl
#include <rolling_max.hpp>

template<typename **Sample**, typename **Weight**>
struct apply
#include <rolling_max.hpp>

Public Types

typedef *hpx*::*util*::detail::rolling_max_impl<Sample> **type**

Header `hpx/statistics/rolling_min.hpp`

namespace boost

namespace accumulators

namespace extract

Variables

extractor<*tag*::*rolling_min*> **const rolling_min** = {}

namespace tag

struct rolling_min : **public** depends_on<rolling_window>
#include <rolling_min.hpp>

struct impl
#include <rolling_min.hpp>

template<typename **Sample**, typename **Weight**>
struct apply
#include <rolling_min.hpp>

Public Types

typedef *hpx*::*util*::detail::rolling_min_impl<Sample> **type**

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.

Header `hpx/string_util/case_conv.hpp`

namespace hpx

namespace string_util

Functions

template<typename **CharT**, class **Traits**, class **Alloc**>
void **to_lower**(*std*::basic_string<*CharT*, *Traits*, *Alloc*> &*s*)

Header `hpx/string_util/classification.hpp`

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
#include <classification.hpp>

Public Functions

bool **operator()**(int *c*) **const**

Header `hpx/string_util/split.hpp`

namespace hpx

namespace string_util

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)

Header hpx/string_util/trim.hpp

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.

Header hpx/condition_variable.hpp

Header hpx/local/barrier.hpp

namespace hpx

Typedefs

using barrier = *lcos*::*local*::cpp20_barrier<OnCompletion>

Header hpx/local/latch.hpp

namespace hpx

Typedefs

using latch = *hpx*::*lcos*::*local*::*cpp20_latch*

Header hpx/mutex.hpp

Header hpx/semaphore.hpp

namespace hpx

Typedefs

using counting_semaphore = *hpx*::*lcos*::*local*::cpp20_counting_semaphore<LeastMaxValue>

using binary_semaphore = *hpx*::*lcos*::*local*::cpp20_binary_semaphore<>

Header hpx/shared_mutex.hpp

Header hpx/stop_token.hpp

Header hpx/synchronization/barrier.hpp

namespace hpx

namespace lcos

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<typename **OnCompletion** = detail::empty_oncompletion>
class cpp20_barrier
#include <barrier.hpp>

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::ptrd

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 Members

mutex_type **mtx_**

local::detail::condition_variable **cond_**

std::ptrdiff_t **expected_**

std::ptrdiff_t **arrived_**

OnCompletion **completion_**

bool **phase_**

Header `hpx/synchronization/channel_mpmc.hpp`

namespace hpx

namespace lcos

namespace local

Typedefs

using channel_mpmc = bounded_channel<T, *hpx*::*lcos*::*local*::*spinlock*>

template<typename **T**, typename **Mutex** = *util*::*spinlock*>
class bounded_channel
#include <channel_mpmc.hpp>

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_**

Header hpx/synchronization/channel_mpsc.hpp

namespace hpx

namespace lcos

namespace local

Typedefs

using channel_mpsc = base_channel_mpsc<T, *hpx*::*lcos*::*local*::*spinlock*>

template<typename **T**, typename **Mutex** = *util*::*spinlock*>
class base_channel_mpsc
#include <channel_mpsc.hpp>

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_**

Header hpx/synchronization/channel_spsc.hpp

namespace hpx

namespace lcos

namespace local

template<typename **T**>
class channel_spsc
#include <channel_spsc.hpp>

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**

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_**

Header hpx/synchronization/condition_variable.hpp

namespace hpx

namespace lcos

namespace local

Enums

enum cv_status
Values:

no_timeout

timeout

error

class condition_variable
#include <condition_variable.hpp>

Public Functions

condition_variable ()

~condition_variable ()

void **notify_one** (*error_code &ec = throws*)

void **notify_all** (*error_code &ec = throws*)

void **wait** (*std*::unique_lock<*mutex*> *&lock*, *error_code &ec = throws*)

template<typename **Predicate**>
void **wait** (*std*::unique_lock<*mutex*> *&lock*, *Predicate pred*, *error_code& = throws*)

cv_status **wait_until**(*std*::unique_lock<*mutex*> &*lock*, *util*::*steady_time_point* **const** &*abs_time*, *error_code* &*ec* = *throws*)

template<typename **Predicate**>
bool **wait_until**(*std*::unique_lock<*mutex*> &*lock*, *util*::*steady_time_point* **const** &*abs_time*, *Predicate pred*, *error_code* &*ec* = *throws*)

cv_status **wait_for**(*std*::unique_lock<*mutex*> &*lock*, *util*::*steady_duration* **const** &*rel_time*, *error_code* &*ec* = *throws*)

template<typename **Predicate**>
bool **wait_for**(*std*::unique_lock<*mutex*> &*lock*, *util*::*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_**

class condition_variable_any
#include <condition_variable.hpp>

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* &*ec* = *throws*)

template<typename **Lock**>
cv_status **wait_until**(*Lock* &*lock*, *util*::*steady_time_point* **const** &*abs_time*, *error_code* &*ec* = *throws*)

template<typename **Lock**, typename **Predicate**>
bool **wait_until**(*Lock* &*lock*, *util*::*steady_time_point* **const** &*abs_time*, *Predicate pred*, *error_code* &*ec* = *throws*)

template<typename **Lock**>
cv_status **wait_for**(*Lock* &*lock*, *util*::*steady_duration* **const** &*rel_time*, *error_code* &*ec* = *throws*)

template<typename **Lock**, typename **Predicate**>

bool **wait_for** (*Lock &lock*, *util::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*, *util::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*, *util::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_**

Header `hpx/synchronization/counting_semaphore.hpp`

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,
#include <counting_semaphore.hpp>

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** ()

Private Types

template<>
using mutex_type = Mutex

template<typename **Mutex** = *hpx*::*lcos*::*local*::*spinlock*>
class cpp20_binary_semaphore : **public** *hpx*::*lcos*::*local*::cpp20_counting_semaphore<1, *Mutex*>
#include <counting_semaphore.hpp>

Public Functions

HPX_NON_COPYABLE (cpp20_binary_semaphore)

cpp20_binary_semaphore (*std*::ptrdiff_t *value* = 1)

~cpp20_binary_semaphore ()

template<*std*::ptrdiff_t **LeastMaxValue** = PTRDIFF_MAX, typename **Mutex** = *hpx*::*lcos*::*local*::*spinlock*>
class cpp20_counting_semaphore
#include <counting_semaphore.hpp>

Public Functions

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** (*util*::*steady_time_point* **const** &*abs_time*)

bool **try_acquire_for** (*util*::*steady_duration* **const** &*rel_time*)

Public Static Functions

static constexpr std::ptrdiff_t() hpx::lcos::local::cpp20_counting_semaphore:

Protected Types

template<>
using mutex_type = Mutex

Protected Attributes

mutex_type **mtx_**

detail::counting_semaphore **sem_**

Header hpx/synchronization/event.hpp

namespace hpx

namespace lcos

namespace local

class event

#include <event.hpp> Event semaphores can be used for synchronizing multiple threads that need to wait for an event to occur. When the event occurs, all threads waiting for the event are woken up.

Public Functions

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

typedef *lcos*::*local*::*spinlock* **mutex_type**

Private Functions

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_**

Header `hpx/synchronization/latch.hpp`

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 singleuse 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

typedef *lcos*::*local*::*spinlock* **mutex_type**

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.

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_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.:`

Header `hpx/synchronization/lock_types.hpp`

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
#include <lock_types.hpp>

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
#include <lock_types.hpp>

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**

Header `hpx/synchronization/mutex.hpp`

namespace hpx

namespace lcos

namespace local

class mutex

#include <mutex.hpp> 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_**

detail::condition_variable **cond_**

class timed_mutex : **private** *hpx*::*lcos*::*local*::*mutex*

#include <mutex.hpp>

Public Functions

HPX_NON_COPYABLE (*timed_mutex*)

timed_mutex (char **const** ***const** *description* = "")

~timed_mutex ()

bool **try_lock_until** (*util*::*steady_time_point* **const** &*abs_time*, char **const** **description*, *error_code* &*ec* = *throws*)

bool **try_lock_until** (*util*::*steady_time_point* **const** &*abs_time*, *error_code* &*ec* = *throws*)

bool **try_lock_for** (*util*::*steady_duration* **const** &*rel_time*, char **const** **description*, *error_code* &*ec* = *throws*)

bool **try_lock_for** (*util*::*steady_duration* **const** &*rel_time*, *error_code* &*ec* = *throws*)

namespace threads

Typedefs

using thread_id_type = *thread_id*

using thread_self = *coroutines*::detail::coroutine_self

Functions

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).

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.

Header `hpx/synchronization/no_mutex.hpp`

namespace hpx

namespace lcos

namespace local

struct no_mutex

#include <no_mutex.hpp>

Public Functions

void **lock**()

bool **try_lock**()

void **unlock**()

Header `hpx/synchronization/once.hpp`

Defines

HPX_ONCE_INIT

namespace hpx

namespace lcos

namespace local

Functions

template<typename **F**, typename ...**Args**>
void **call_once**(*once_flag &flag*, *F &&f*, *Args&&... args*)

struct once_flag
#include <once.hpp>

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*)

Header hpx/synchronization/recursive_mutex.hpp

namespace hpx

namespace lcos

namespace local

Typedefs

using recursive_mutex = detail::recursive_mutex_impl<>

Header hpx/synchronization/shared_mutex.hpp

namespace hpx

namespace lcos

namespace local

Typedefs

typedef detail::shared_mutex **shared_mutex**

Header hpx/synchronization/sliding_semaphore.hpp

namespace hpx

namespace lcos

namespace local

Typedefs

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()*.
- `lower_limit`: [in] The initial lower limit.

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_**

Header hpx/synchronization/spinlock.hpp

namespace hpx

namespace lcos

namespace local

struct spinlock
#include <spinlock.hpp>

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_**

Header `hpx/synchronization/spinlock_no_backoff.hpp`

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

std::atomic<bool> **v_**

Header `hpx/synchronization/spinlock_pool.hpp`

namespace hpx

namespace lcos

namespace local

template<typename **Tag**, *std*::size_t **N** = HPX_HAVE_SPINLOCK_POOL_NUM>
class spinlock_pool
#include <spinlock_pool.hpp>

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
#include <spinlock_pool.hpp>

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_**

Header `hpx/synchronization/stop_token.hpp`

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
#include <stop_token.hpp>

Public Functions

nostopstate_t()

template<typename **Callback**>
class stop_callback : **private** *hpx*::detail::stop_callback_base
#include <stop_token.hpp>

Public Types

template<>
using callback_type = Callback

Public Functions

template<typename **CB**, typename **Enable** = **typename** *std*::enable_if<*std*::is_constructible<Callback, *CB*>::value>::typ
stop_callback(*stop_token* **const** *&st*, *CB &&cb*)

template<typename **CB**, typename **Enable** = **typename** *std*::enable_if<*std*::is_constructible<Callback, *CB*>::value>::typ
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
#include <stop_token.hpp>

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_**

Friends

HPX_NODISCARD friend bool operator==(stop_source const & lhs, stop_source const &

HPX_NODISCARD friend bool operator!=(stop_source const & lhs, stop_source const &

class stop_token

#include <stop_token.hpp>

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 & r

HPX_NODISCARD friend bool operator!=(stop_token const & lhs, stop_token const & r

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.

Header `hpx/modules/testing.hpp`

Defines

HPX_TEST(...)

HPX_TEST_(...)

HPX_TEST_1(*expr*)

HPX_TEST_2(*strm*, *expr*)

HPX_TEST_IMPL(*fixture*, *expr*)

HPX_TEST_MSG(...)

HPX_TEST_MSG_(...)

HPX_TEST_MSG_2(*expr*, *msg*)

HPX_TEST_MSG_3(*strm*, *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(*strm*, *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(*strm*, *expr1*, *expr2*)

HPX_TEST_NEQ_IMPL(*fixture*, *expr1*, *expr2*)

HPX_TEST_LT(...)

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*)

HPX_SANITY_RANGE_4 (*strm*, *expr1*, *expr2*, *expr3*)

HPX_SANITY_RANGE_IMPL (*fixture*, *expr1*, *expr2*, *expr3*)

HPX_SANITY_EQ_MSG (...)

HPX_SANITY_EQ_MSG_ (...)

HPX_SANITY_EQ_MSG_3 (*expr1*, *expr2*, *msg*)

HPX_SANITY_EQ_MSG_4 (*strm*, *expr1*, *expr2*, *msg*)

HPX_SANITY_EQ_MSG_IMPL (*fixture*, *expr1*, *expr2*, *msg*)

HPX_TEST_THROW (...)

HPX_TEST_THROW_ (...)

HPX_TEST_THROW_2 (*expression*, *exception*)

HPX_TEST_THROW_3 (*strm*, *expression*, *exception*)

HPX_TEST_THROW_IMPL (*fixture*, *expression*, *exception*)

namespace hpx

namespace util

Typedefs

using test_failure_handler_type = *function_nonser*<void ()>

Enums

enum counter_type

Values:

counter_sanity

counter_test

Functions

void **set_test_failure_handler** (*test_failure_handler_type f*)

int **report_errors** (*std*::ostream &*stream* = *std*::cerr)

void **print_cdash_timing** (**const** char **name*, double *time*)

void **print_cdash_timing** (**const** char **name*, *std*::uint64_t *time*)

thread_executors

The contents of this module can be included with the header `hpx/modules/thread_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/thread_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.

Header `hpx/execution/executors/default_executor.hpp`

namespace hpx

namespace parallel

namespace execution

Typedefs

using default_executor = *parallel_executor*

Header `hpx/execution/executors/this_thread_executors.hpp`

Header `hpx/execution/executors/thread_pool_executors.hpp`

Header `hpx/execution/executors/thread_pool_os_executors.hpp`

Header `hpx/thread_executors/current_executor.hpp`

Header `hpx/thread_executors/default_executor.hpp`

Header `hpx/thread_executors/embedded_thread_pool_executors.hpp`

Header `hpx/thread_executors/executors.hpp`

Header `hpx/thread_executors/guided_pool_executor.hpp`

Header `hpx/thread_executors/limiting_executor.hpp`

Header `hpx/thread_executors/manage_thread_executor.hpp`

Header `hpx/thread_executors/resource_manager.hpp`

Header `hpx/thread_executors/this_thread_executors.hpp`

Header `hpx/thread_executors/thread_execution.hpp`

Header `hpx/thread_executors/thread_execution_information.hpp`

namespace hpx

Functions

std::size_t **get_os_thread_count** (*threads*::executor **const** &*exec*)

namespace threads

Functions

template<typename **Executor**, typename **Parameters**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value, *std*::size_t>::type **processing_units_count** (*Para* *Executor* &&*e*

template<typename **Executor**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value, bool>::type **has_pending_closures** (*Executor* &&*exec*)

template<typename **Executor**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value, *threads*::mask_cref_type>::type **get_pu_mask** (*Execut* &&*exe* *thread* &*topo*, *std*::*siz* *thread*

template<typename **Executor**, typename **Mode**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value>::type **set_scheduler_mode** (*Executor* &&*exec*, *Mode* *mode*)

Header hpx/thread_executors/thread_executor.hpp

Header hpx/thread_executors/thread_pool_attached_executors.hpp

Header hpx/thread_executors/thread_pool_os_executors.hpp

Header hpx/thread_executors/thread_timed_execution.hpp

namespace hpx

namespace threads

Functions

template<typename **Executor**, typename **F**, typename ...**Ts**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value>::type **post_at** (*Executor* &&*exec*, *hpx*::*util*::*steady_time_point* **const** &*abs_time*, *F* &&*f*, *Ts*&&... *ts*)

template<typename **Executor**, typename **F**, typename ...**Ts**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value>::type **post_after** (*Executor* &&*exec*, *hpx*::*util*::*steady_duration* **const** &*rel_time*, *F* &&*f*, *Ts*&&... *ts*)

template<typename **Executor**, typename **F**, typename ...**Ts**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value, *hpx*::future<**typename** *hpx*::*util*::detail::invoke_deferred

template<typename **Executor**, typename **F**, typename ...**Ts**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value, *hpx*::future<**typename** *hpx*::*util*::detail::invoke_deferred

template<typename **Executor**, typename **F**, typename ...**Ts**>
std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value, **typename** *hpx*::*util*::detail::invoke_deferred_result<*F*, *T*

template<typename **Executor**, typename **F**, typename ...**Ts**>

std::enable_if<*hpx*::*traits*::is_threads_executor<*Executor*>::value, **typename** *hpx*::*util*::detail::invoke_deferred_result<*F*, *T*

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.

Header `hpx/thread_pools/scheduled_thread_pool.hpp`

Header `hpx/thread_pools/scheduled_thread_pool_impl.hpp`

Header `hpx/thread_pools/scheduling_loop.hpp`

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.

Header `hpx/thread_support/assert_owns_lock.hpp`

Defines

HPX_ASSERT_OWNS_LOCK(*l*)

HPX_ASSERT_DOESNT_OWN_LOCK(*l*)

Header `hpx/thread_support/atomic_count.hpp`

namespace hpx

namespace util

class atomic_count

#include <atomic_count.hpp>

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_**

Header `hpx/thread_support/set_thread_name.hpp`

namespace hpx

namespace util

Functions

void **set_thread_name**(char **const***)

Header `hpx/thread_support/thread_specific_ptr.hpp`

Defines

HPX_EXPORT_THREAD_SPECIFIC_PTR

Header `hpx/thread_support/unlock_guard.hpp`

namespace hpx

namespace util

template<typename **Mutex**>
class unlock_guard
#include <unlock_guard.hpp>

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.

Header `hpx/thread.hpp`

Header `hpx/threading/jthread.hpp`

namespace hpx

Functions

void **swap** (*jthread* &*lhs*, *jthread* &*rhs*)

class jthread
#include <jthread.hpp>

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*::dec
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 **ssource_**

hpx::*thread* **thread_** = {}

Private Static Functions

template<typename **F**, typename ...**Ts**>
static void **invoke**(*std*::false_type, *F* &&*f*, *stop_token* &&*st*, *Ts*&&... *ts*)

template<typename **F**, typename ...**Ts**>
static void **invoke**(*std*::true_type, *F* &&*f*, *stop_token* &&*st*, *Ts*&&... *ts*)

Header hpx/threading/thread.hpp

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

#include <thread.hpp>

Public Types

typedef *threads*::*thread_id_type* **native_handle_type**

Public Functions

thread ()

template<typename **F**, typename **Enable** = **typename** *std*::enable_if<!*std*::is_same<**typename** *hpx*::*util*::decay<*F*>::ty
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()

static void **interrupt**(*id*, bool *flag* = true)

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_type* **id_**

Private Static Functions

static *threads*::thread_result_type **thread_function_nullary**(*util*::*unique_function_nonser*<void()> **const** &*func*)

class id
#include <thread.hpp>

Public Functions

id()

id(*threads*::*thread_id_type* **const** &*i*)

id(*threads*::*thread_id_type* &&*i*)

threads::*thread_id_type* **const** &**native_handle**() **const**

Private Members

threads::*thread_id_type* **id_**

Friends

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*)

template<typename **Char**, typename **Traits**>
std::basic_ostream<*Char*, *Traits*> &**operator<<**(*std*::basic_ostream<*Char*, *Traits*> &*out*, thread::id **const** &*id*)

namespace this_thread

Functions

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**(*util*::*steady_time_point* **const** &*abs_time*)

void **sleep_for**(*util*::*steady_duration* **const** &*rel_time*)

std::size_t **get_thread_data**()

std::size_t **set_thread_data**(*std*::size_t)

class disable_interruption

#include <thread.hpp>

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
#include <thread.hpp>

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_**

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.

Header `hpx/threading_base/annotated_function.hpp`

namespace hpx

namespace util

Functions

template<typename **F**>
F &&**annotated_function**(*F* &&*f*, char **const*** = nullptr)
Given a function as an argument, the user can *annotate_function* as well. Annotating includes setting the thread description per thread id.

Parameters
- `function`:

struct annotate_function
#include <annotated_function.hpp>

Public Functions

HPX_NON_COPYABLE (*annotate_function*)

annotate_function (char **const***)

template<typename **F**>
annotate_function (*F*&&)

~annotate_function ()

Header hpx/threading_base/callback_notifier.hpp

namespace hpx

namespace threads

namespace policies

class callback_notifier
#include <callback_notifier.hpp>

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_**

Header `hpx/threading_base/create_thread.hpp`

Header `hpx/threading_base/create_work.hpp`

Header `hpx/threading_base/execution_agent.hpp`

namespace hpx

namespace threads

struct execution_agent : **public** agent_base
#include <execution_agent.hpp>

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*::*util*::*steady_duration* **const** &*sleep_duration*, char **const** **desc*)

void **sleep_until** (*hpx*::*util*::*steady_time_point* **const** &*sleep_time*, char **const** **desc*)

Private Functions

hpx::*threads*::*thread_state_ex_enum* **do_yield** (char **const** **desc*, *threads*::*thread_state_enum state*)

void **do_resume** (char **const** **desc*, *hpx*::*threads*::*thread_state_ex_enum statex*)

Private Members

coroutines::detail::coroutine_stackful_self **self_**

execution_context **context_**

struct execution_context : **public** context_base
#include <execution_agent.hpp>

Public Functions

hpx::*execution_base*::*resource_base* **const** &**resource**() **const**

Public Members

hpx::*execution_base*::*resource_base* **resource_**

Header `hpx/threading_base/external_timer.hpp`

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* **const**&)

std::shared_ptr<task_wrapper> **update_task**(*std*::shared_ptr<task_wrapper>, *thread_description* **const**&)

struct scoped_timer
#include <external_timer.hpp>

Public Functions

scoped_timer(*std*::shared_ptr<task_wrapper>)

~scoped_timer()

void **stop**(void)

void **yield**(void)

Header hpx/threading_base/network_background_callback.hpp

Header hpx/threading_base/print.hpp

Header hpx/threading_base/register_thread.hpp

namespace hpx

namespace threads

Functions

template<typename **F**>
thread_function_type **make_thread_function** (*F &&f*)

template<typename **F**>
thread_function_type **make_thread_function_nullary** (*F &&f*)

threads::*thread_id_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.

threads::*thread_id_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.

void **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::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.

void **register_work**(*threads*::*thread_init_data* &*data*, *error_code* &*ec* = *throws*)

Create a new work item using the given data on the same thread pool as the calling thread, or on the default thread pool if not on an HPX thread.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- `data`: [in] The data to use for creating the thread.
- `ec`: [in,out] This represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

Exceptions
- `invalid_status`: if the runtime system has not been started yet.

Header `hpx/threading_base/scheduler_base.hpp`

namespace hpx

namespace threads

namespace policies

struct scheduler_base

#include <scheduler_base.hpp> The *scheduler_base* defines the interface to be implemented by all scheduler policies

Public Types

typedef *std*::mutex **pu_mutex_type**

using polling_function_ptr = void (*) ()

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**()

void **set_parent_pool**(*threads*::*thread_pool_base* **p*) *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)) **const** = 0 virtual *std*::int64_t **get_thread_count** (*thread_state_enum 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_state_enum 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_type* **id*, *error_code* &*ec*) = 0

virtual bool **get_next_thread** (*std*::size_t *num_thread*, bool *running*, *threads*::*thread_data* *&*thrd*, bool *enable_stealing*) = 0

virtual void **schedule_thread** (*threads*::*thread_data* **thrd*, *threads*::*thread_schedule_hint schedulehint*, bool *allow_fallback* = false, *thread_priority priority* = *thread_priority_normal*) = 0

virtual void **schedule_thread_last** (*threads*::*thread_data* **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_function** (*polling_function_ptr mpi_func*)

void **clear_mpi_polling_function** ()

void **set_cuda_polling_function** (*polling_function_ptr cuda_func*) void **clear_cuda_polling_function**()

void **custom_polling_function**() **const**

Public Static Functions

static void **null_polling_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_**

Header `hpx/threading_base/scheduler_mode.hpp`

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 processing 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 schedulersthat 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*
This option represents the default mode.

all_flags = *do_background_work* | *reduce_thread_priority* | *delay_exit* | *fast_idle_mode* | *enable_elasticity* | *en*
This enables all available options.

Header hpx/threading_base/scheduler_state.hpp

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*

Header hpx/threading_base/set_thread_state.hpp

Header hpx/threading_base/thread_data.hpp

namespace hpx

namespace threads

Functions

constexpr *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).

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_id_type **get_parent_id**()
The function *get_parent_id* returns the HPX thread id of the current thread's parent (or zero if the current thread is not a HPX thread).

Note This function will return a meaningful value only if the code was compiled with HPX_HAVE_THREAD_PARENT_REFERENCE being defined.

std::size_t **get_parent_phase**()
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
#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 mutex_type = *util*::spinlock_pool<*tag*>

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_state_enum state*, *thread_state_ex_enum state_ex* = *wait_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_state_enum newstate*, thread_state &*prev_state*, thread_state &*new_tagged_state*, *std*::memory_order *exchange_order* = *std*::memory_order_seq_cst)

bool **restore_state**(thread_state *new_state*, thread_state *old_state*, *std*::memory_order *load_order* = *std*::memory_order_relaxed, *std*::memory_order *load_exchange* = *std*::memory_order_seq_cst)

The restore_state function changes the state of this thread instance depending on its current state. It will change the state atomically only if the current state is still the same as passed as the second parameter. Otherwise it won't touch the thread state of this instance.

Note This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager. Moreover, changing the thread state using this function does not change its scheduling status. It only sets the thread's status word. To change the thread's scheduling status *threadmanager::set_state* should be used.

Return This function returns *true* if the state has been changed successfully

Parameters

- `newstate`: [in] The new state to be set for the thread.
- `oldstate`: [in] The old state of the thread which still has to be the current state.

bool **restore_state**(*thread_state_enum new_state*, *thread_state_ex_enum state_ex*, thread_state *old_state*, *std*::memory_order *load_exchange* = *std*::memory_order_seq_cst)

constexpr *naming*::*address_type* **get_component_id**() **const**
Return the id of the component this thread is running in.

util::*thread_description* **get_description**() **const**

util::*thread_description* **set_description**(*util*::*thread_description*)

util::*thread_description* **get_lco_description**() **const**

util::*thread_description* **set_lco_description**(*util*::*thread_description*)

constexpr *std*::uint32_t **get_parent_locality_id**() **const**
Return the locality of the parent thread.

constexpr *thread_id_type* **get_parent_thread_id**() **const**
Return the thread id of the parent thread.

constexpr *std*::size_t **get_parent_thread_phase**() **const**
Return the phase of the parent thread.

constexpr *util*::backtrace **const** ***get_backtrace**() **const**

util::backtrace **const** ***set_backtrace**(*util*::backtrace **const***)

constexpr *thread_priority* **get_priority**() **const**

void **set_priority**(*thread_priority priority*)

bool **interruption_requested**() **const**

bool **interruption_enabled**() **const**

bool **set_interruption_enabled**(bool *enable*)

void **interrupt**(bool *flag* = true)

bool **interruption_point**(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**()

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 **rebind**(*thread_init_data* &*init_data*) = 0

thread_data(*thread_init_data* &*init_data*, void **queue*, *std*::ptrdiff_t *stacksize*, bool *is_stackless* = false)

virtual ~thread_data()

virtual void **destroy**() = 0

Public Members

bool **is_stackless_**

Protected Functions

thread_state_ex_enum **set_state_ex**(*thread_state_ex_enum new_state*)

The set_state function changes the extended state of this thread instance.

Note This function will be seldom used directly. Most of the time the state of a thread will have to be changed using the threadmanager.

Parameters

- `newstate`: [in] The new extended state to be set for the thread.

void **rebind_base**(*thread_init_data* &*init_data*)

Private Members

std::atomic<thread_state> **current_state_**

thread_priority **priority_**

bool **requested_interrupt_**

bool **enabled_interrupt_**

bool **ran_exit_funcs_**

std::forward_list<*util*::*function_nonser*<void()>> **exit_funcs_**

policies::*scheduler_base* ***scheduler_base_** *std*::size_t **last_worker_thread_num_**

std::ptrdiff_t **stacksize_**

thread_stacksize **stacksize_enum_**

void ***queue_**

Header hpx/threading_base/thread_data_stackful.hpp

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 *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

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 **rebind**(*thread_init_data* &*init_data*)

thread_data_stackful(*thread_init_data* &*init_data*, void **queue*, *std*::ptrdiff_t *stacksize*)

~thread_data_stackful()

void **destroy**()

Public Static Functions

thread_data ***create**(*thread_init_data* &*init_data*, void **queue*, *std*::ptrdiff_t *stacksize*)

Private Functions

thread_data ***this_**()

Private Members

coroutine_type **coroutine_**

execution_agent **agent_**

Private Static Attributes

util::*internal_allocator*<*thread_data_stackful*> **thread_alloc_**

Header `hpx/threading_base/thread_data_stackless.hpp`

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 **rebind**(*thread_init_data* &*init_data*)

thread_data_stackless(*thread_init_data* &*init_data*, void **queue*, *std*::ptrdiff_t *stacksize*)

~thread_data_stackless()

void **destroy**()

Public Static Functions

thread_data ***create**(*thread_init_data* &*init_data*, void **queue*, *std*::ptrdiff_t *stacksize*)

Private Functions

thread_data ***this_**()

Private Members

stackless_coroutine_type **coroutine_**

Private Static Attributes

util::*internal_allocator*<*thread_data_stackless*> **thread_alloc_**

Header hpx/threading_base/thread_description.hpp

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
#include <thread_description.hpp>

Public Types

enum data_type
Values:

data_type_description = 0

data_type_address = 1

Public Functions

thread_description()

thread_description(char **const***)

template<typename **F**, typename = **typename** *std*::enable_if<!*std*::is_same<*F*, *thread_description*>::value && !*traits*
thread_description(*F* **const**&, char **const*** = nullptr)

template<typename **Action**, typename = **typename** *std*::enable_if<*traits*::is_action<*Action*>::value>::type>
thread_description(*Action*, char **const*** = nullptr)

data_type **kind**() **const**

char **const** ***get_description**() **const**

std::size_t **get_address**() **const**

operator bool() **const**

bool **valid**() **const**

Private Functions

void **init_from_alternative_name**(char **const** **altname*)

Header hpx/threading_base/thread_helpers.hpp

namespace hpx

namespace this_thread

Functions

threads::*thread_state_ex_enum* **suspend**(*threads*::*thread_state_enum state*, *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 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_state_ex_enum* **suspend**(*threads*::*thread_state_enum state* = *threads*::*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_state_ex_enum* **suspend**(*util*::*steady_time_point* **const** &*abs_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 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_state_ex_enum* **suspend**(*util*::*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_state_ex_enum* **suspend**(*util*::*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 threads 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*.

threads::*thread_state_ex_enum* **suspend**(*util*::*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 threads 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*.

threads::*thread_state_ex_enum* **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 threads 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*.

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 *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 threads

Functions

thread_state **set_thread_state**(*thread_id_type* **const** &*id*, *thread_state_enum state* = *pending*, *thread_state_ex_enum stateex* = *wait_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_type **set_thread_state**(*thread_id_type* **const** &*id*, *util*::*steady_time_point* **const** &*abs_time*, *std*::atomic<bool> **started*, *thread_state_enum state* = *pending*, *thread_state_ex_enum stateex* = *wait_timeout*, *thread_priority priority* = *thread_priority_normal*, bool *retry_on_active* = true, *error_code* &*ec* = *throws*)

Set the thread state of the *thread* referenced by the thread_id *id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (at the given time)

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.

thread_id_type **set_thread_state**(*thread_id_type* **const** &*id*, *util*::*steady_time_point* **const** &*abs_time*, *thread_state_enum state* = *pending*, *thread_state_ex_enum stateex* = *wait_timeout*, *thread_priority priority* = *thread_priority_normal*, bool *retry_on_active* = true, *error_code*& = *throws*)

thread_id_type **set_thread_state**(*thread_id_type* **const** &*id*, *util*::*steady_duration* **const** &*rel_time*, *thread_state_enum state* = *pending*, *thread_state_ex_enum stateex* = *wait_timeout*, *thread_priority priority* = *thread_priority_normal*, bool *retry_on_active* = true, *error_code* &*ec* = *throws*)

Set the thread state of the *thread* referenced by the thread_id *id*.

Set a timer to set the state of the given *thread* to the given new value after it expired (after the given duration)

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.
- `rel_time`: [in] Time duration after which the new thread should be run
- `state`: [in] The new state to be set for the thread referenced by the *id* parameter.
- `stateex`: [in] The new extended state to be set for the thread referenced by the *id* parameter.
- `priority`:
- `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_state **get_thread_state**(*thread_id_type* **const** &*id*, *error_code* &*ec* = *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.
- `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 **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 **interrupt_thread**(*thread_id_type* **const** &*id*, *error_code* &*ec* = *throws*)

void **interruption_point**(*thread_id_type* **const** &*id*, *error_code* &*ec* = *throws*)
Interrupt the current thread at this point if it was canceled. This will throw a thread_interrupted exception, which will cancel the thread.

Note As long as *ec* is not pre-initialized to *hpx::throws* this function doesn't throw but returns the result code using the parameter *ec*. Otherwise it throws an instance of hpx::exception.

Parameters
- `id`: [in] The thread id of the thread which should be interrupted.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

threads::*thread_priority* **get_thread_priority**(*thread_id_type* **const** &*id*, *error_code* &*ec* = *throws*)
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*.

Header `hpx/threading_base/thread_init_data.hpp`

namespace hpx

namespace threads

Functions

std::ptrdiff_t **get_default_stack_size**()

std::ptrdiff_t **get_stack_size**(*thread_stacksize*)

class thread_init_data
#include <thread_init_data.hpp>

Public Functions

thread_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_state_enum* *initial_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_state_enum **initial_state**

bool **run_now**

policies::*scheduler_base* ***scheduler_base**

Header hpx/threading_base/thread_num_tss.hpp

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.

std::size_t **`get_thread_pool_num`**`()`

Return the number of the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the thread pool which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the number of thread pools started by the runtime. It will return -1 if the current thread pool is not a known thread pool or if the runtime is not in running state.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

std::size_t **`get_thread_pool_num`**`(`*error_code &ec*`)`

Return the number of the current thread pool the current HPX-thread is executed with.

This function returns the zero based index of the thread pool which executes the current HPX-thread.

Note The returned value is zero based and its maximum value is smaller than the number of thread pools started by the runtime. It will return -1 if the current thread pool is not a known thread pool or if the runtime is not in running state.

Note This function needs to be executed on a HPX-thread. It will fail otherwise (it will return -1).

Parameters

- `ec`: [in,out] this represents the error status on exit.

Header `hpx/threading_base/thread_pool_base.hpp`

`namespace hpx`

`namespace threads`

`struct executor_statistics`

#include <thread_pool_base.hpp> Data structure which stores statistics collected by an executor instance.

Public Functions

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.

struct thread_pool_init_parameters

#include <thread_pool_base.hpp>

Public Functions

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)

Public Members

std::string **const** &**name_**

std::size_t **index_**

policies::*scheduler_mode* **mode_**

std::size_t **num_threads_**

std::size_t **thread_offset_**

hpx::*threads*::*policies*::*callback_notifier* &**notifier_**

hpx::*threads*::*policies*::detail::affinity_data **const** &**affinity_data_**

hpx::*threads*::detail::network_background_callback_type **const** &**network_background_callback_**

std::size_t **max_background_threads_**

std::size_t **max_idle_loop_count_**

std::size_t **max_busy_loop_count_**

Header `hpx/threading_base/thread_queue_init_parameters.hpp`

namespace hpx

namespace threads

namespace policies

struct thread_queue_init_parameters

#include <thread_queue_init_parameters.hpp>

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_PE *std*::int64_t *min_tasks_to_steal_staged* = *std*::int64_t(HPX_THREAD_QUEUE_MIN_TASKS_TO_STEAL_STA *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_THREA 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_**

double **max_idle_backoff_time_**

std::ptrdiff_t **const small_stacksize_**

std::ptrdiff_t **const medium_stacksize_**

std::ptrdiff_t **const large_stacksize_**

std::ptrdiff_t **const huge_stacksize_**

std::ptrdiff_t **const nostack_stacksize_**

Header hpx/threading_base/thread_specific_ptr.hpp

namespace hpx

namespace threads

template<typename **T**>
class thread_specific_ptr
#include <thread_specific_ptr.hpp>

Public Types

typedef T **element_type**

Public Functions

thread_specific_ptr()

thread_specific_ptr(void (**func_*)) T*

~thread_specific_ptr()

T ***get**() **const**

T ***operator->**() **const**

T &**operator***() **const**

T ***release**()

void **reset**(T **new_value* = nullptr)

Private Types

typedef *coroutines*::detail::tss_cleanup_function **cleanup_function**

Private Functions

thread_specific_ptr(*thread_specific_ptr&*)

thread_specific_ptr &**operator=**(thread_specific_ptr&)

Private Members

std::shared_ptr<*cleanup_function*> **cleanup_**

struct delete_data : **public** tss_cleanup_function

Public Functions

template<>
void **operator()** (void **data*)

struct run_custom_cleanup_function : **public** tss_cleanup_function

Public Functions

template<>
run_custom_cleanup_function (void (**cleanup_function_*)) T*

template<>
void **operator()** (void **data*)

Public Members

template<>
void (***cleanup_function**) (T*)

Header `hpx/threading_base/threading_base_fwd.hpp`

template<>
struct hash<::*hpx*::*threads*::*thread_id*>
#include <threading_base_fwd.hpp>

Public Functions

std::size_t **operator()** (::*hpx*::*threads*::*thread_id* **const** &*v*) **const**

namespace std

template<>
struct hash<::*hpx*::*threads*::*thread_id*>
#include <threading_base_fwd.hpp>

Public Functions

std::size_t **operator()**(::*hpx*::*threads*::*thread_id* **const** &*v*) **const**

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.

Header `hpx/modules/threadmanager.hpp`

namespace hpx

namespace threads

class threadmanager

#include <threadmanager.hpp> The *thread-manager* class is the central instance of management for all (non-depleted) threads

Public Types

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(*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**(*std*::size_t *pool_index*) **const**

void **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_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`:

void **suspend**()

void **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_state_enum 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_state_enum 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_remove_scheduler_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::size_t **shrink_pool**(*std*::string **const** &*pool_name*)

std::size_t **expand_pool**(*std*::string **const** &*pool_name*)

std::int64_t **get_queue_length**(bool *reset*)

std::int64_t **get_cumulative_duration**(bool *reset*)

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_**

std::size_t **num_threads_**

std::vector<pool_id_type> **threads_lookup_**

pool_vector **pools_**

notification_policy_type &**notifier_**

detail::network_background_callback_type **network_background_callback_**

Header `hpx/threadmanager/threadmanager_fwd.hpp`

timed_execution

The contents of this module can be included with the header `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 *strongly* suggest only including the module header `hpx/modules/timed_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.

Header `hpx/timed_execution/timed_execution.hpp`

Header `hpx/timed_execution/timed_execution_fwd.hpp`

Header `hpx/timed_execution/timed_executors.hpp`

namespace hpx

namespace parallel

namespace execution

Typedefs

using sequenced_timed_executor = timed_executor<*execution*::*sequenced_executor*>

using parallel_timed_executor = timed_executor<*execution*::*parallel_executor*>

template<typename **BaseExecutor**>
struct timed_executor
#include <timed_executors.hpp>

Public Types

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

timed_executor(*hpx*::*util*::*steady_time_point* **const** &*abs_time*)

timed_executor(*hpx*::*util*::*steady_duration* **const** &*rel_time*)

template<typename **Executor**>
timed_executor(*Executor* &&*exec*, *hpx*::*util*::*steady_time_point* **const** &*abs_time*)

template<typename **Executor**>
timed_executor(*Executor* &&*exec*, *hpx*::*util*::*steady_duration* **const** &*rel_time*)

template<typename **F**, typename ...**Ts**>
hpx::*util*::detail::invoke_deferred_result<*F*, *Ts*...>::type **sync_execute**(*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**>
void **post**(*F* &&*f*, *Ts*&&... *ts*)

Public Members

BaseExecutor **exec_**

std::chrono::steady_clock::time_point **execute_at_**

Header `hpx/timed_execution/traits/is_timed_executor.hpp`

namespace hpx

namespace parallel

namespace execution

Typedefs

using is_timed_executor_t = **typename** is_timed_executor<T>::type

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.

Header `hpx/timing/high_resolution_clock.hpp`

namespace hpx

namespace util

struct high_resolution_clock
#include <high_resolution_clock.hpp>

Public Static Functions

static *std*::uint64_t **now**()

static std::uint64_t() hpx::util::high_resolution_clock::min()

static std::uint64_t() hpx::util::high_resolution_clock::max()

Header `hpx/timing/high_resolution_timer.hpp`

namespace hpx

namespace util

class high_resolution_timer
#include <high_resolution_timer.hpp>

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_**

Header `hpx/timing/scoped_timer.hpp`

namespace hpx

namespace util

template<typename **T**>
struct scoped_timer
#include <scoped_timer.hpp>

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_**

Header hpx/timing/steady_clock.hpp

namespace hpx

namespace util

class steady_duration

#include <steady_clock.hpp>

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

#include <steady_clock.hpp>

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**

Header `hpx/timing/tick_counter.hpp`

namespace hpx

namespace util

class tick_counter
#include <tick_counter.hpp>

Public Functions

tick_counter (*std*::uint64_t &*output*)

~tick_counter ()

Protected Static Functions

static *std*::uint64_t **take_time_stamp** ()

Private Members

std::uint64_t **const 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.

Header `hpx/topology/cpu_mask.hpp`

Header `hpx/topology/topology.hpp`

namespace hpx

namespace threads

Typedefs

using hwloc_bitmap_ptr = *std*::shared_ptr<*hpx_hwloc_bitmap_wrapper*>

Enums

enum hpx_hwloc_membind_policy

Please see hwloc documentation for the corresponding enums HWLOC_MEMBIND_XXX.

Values:

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

topology &**create_topology**()

HPX_NODISCARD unsigned int hpx::threads::hardware_concurrency()

std::size_t **get_memory_page_size**()

struct hpx_hwloc_bitmap_wrapper

#include <topology.hpp>

Public Functions

HPX_NON_COPYABLE(*hpx_hwloc_bitmap_wrapper*)

hpx_hwloc_bitmap_wrapper()

hpx_hwloc_bitmap_wrapper(void **bmp*)

~hpx_hwloc_bitmap_wrapper()

void **reset**(hwloc_bitmap_t *bmp*)

operator bool() **const**

hwloc_bitmap_t **get_bmp**() **const**

Private Members

hwloc_bitmap_t **bmp_**

Friends

std::ostream &**operator<<**(*std*::ostream &*os*, hpx_hwloc_bitmap_wrapper **const** **bmp*)

struct topology

#include <topology.hpp>

Public Functions

topology()

~topology()

std::size_t **get_socket_number**(*std*::size_t *num_thread*, *error_code*& = *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.

std::size_t **get_numa_node_number**(*std*::size_t *num_thread*, *error_code*& = *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.

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.

mask_type **get_service_affinity_mask**(mask_cref_type *used_processing_units*, *error_code &ec = throws*) **const**

Return a bit mask where each set bit corresponds to a processing unit available to the service threads in the application.

Parameters
- `used_processing_units`: [in] This is the mask of processing units which are not available for service threads.
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

mask_cref_type **get_socket_affinity_mask**(*std*::size_t *num_thread*, *error_code &ec = throws*) **const**

Return a bit mask where each set bit corresponds to a processing unit available to the given thread inside the socket it is running on.

Parameters
- `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 **const** *lva, *error_code* &*ec* = *throws*) **const**
Return a bit mask where each set bit corresponds to a processing unit co-located with the memory the given address is currently allocated on.

Parameters
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

void **print_affinity_mask**(*std*::ostream &*os*, *std*::size_t *num_thread*, mask_cref_type *m*, **const** *std*::string &*pool_name*) **const**
Prints the.

Parameters
- `m`: to os in a human readable form

bool **reduce_thread_priority**(*error_code* &*ec* = *throws*) **const**
Reduce thread priority of the current thread.

Parameters
- `ec`: [in,out] this represents the error status on exit, if this is pre-initialized to *hpx::throws* the function will throw on error instead.

std::size_t **get_number_of_sockets**() **const**
Return the number of available NUMA domains.

std::size_t **get_number_of_numa_nodes**() **const**
Return the number of available NUMA domains.

std::size_t **get_number_of_cores**() **const**
Return the number of available cores.

std::size_t **get_number_of_pus**() **const**
Return the number of available hardware processing units.

std::size_t **get_number_of_numa_node_cores**(*std*::size_t *numa*) **const**
Return number of cores in given numa domain.

std::size_t **get_number_of_numa_node_pus**(*std*::size_t *numa*) **const**
Return number of processing units in a given numa domain.

std::size_t **get_number_of_socket_pus**(*std*::size_t *socket*) **const**
Return number of processing units in a given socket.

std::size_t **get_number_of_core_pus**(*std*::size_t *core*) **const**
Return number of processing units in given core.

std::size_t **get_number_of_socket_cores**(*std*::size_t *socket*) **const**
Return number of cores units in given socket.

std::size_t **get_core_number**(*std*::size_t *num_thread*, *error_code*& = *throws*) **const**

std::size_t **get_pu_number**(*std*::size_t *num_core*, *std*::size_t *num_pu*, *error_code* &*ec* = *throws*) **const**

mask_type **get_cpubind_mask**(*error_code* &*ec* = *throws*) **const** mask_type **get_cpubind_mask**(*std*::thread &*handle*, *error_code* &*ec* = *throws*) **const**

hwloc_bitmap_ptr **cpuset_to_nodeset**(mask_cref_type *cpuset*) **const**
convert a cpu mask into a numa node mask in hwloc bitmap form

void **write_to_log**() **const**

void ***allocate**(*std*::size_t *len*) **const**
This is equivalent to malloc(), except that it tries to allocate page-aligned memory from the OS.

void ***allocate_membind**(*std*::size_t *len*, *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_**

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_**

const *std*::size_t **pu_offset** = 0

const *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 `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 `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.

Header `hpx/type_support/always_void.hpp`

namespace hpx

namespace util

template<typename ...**T**>
struct always_void
#include <always_void.hpp>

Public Types

typedef void **type**

Header `hpx/type_support/decay.hpp`

Header `hpx/type_support/detected.hpp`

namespace hpx

namespace util

Typedefs

using is_detected = **typename** detail::detector<*nonesuch*, void, Op, Args...>::value_t

using detected_t = **typename** detail::detector<*nonesuch*, void, Op, Args...>::type

using detected_or = detail::detector<Default, void, Op, Args...>

using detected_or_t = **typename** *detected_or*<Default, Op, Args...>::type

using is_detected_exact = *std*::is_same<Expected, *detected_t*<Op, Args...>>

using is_detected_convertible = *std*::is_convertible<*detected_t*<Op, Args...>, To>

struct nonesuch
#include <detected.hpp>

Public Functions

nonesuch()

~nonesuch()

nonesuch(*nonesuch* **const**&)

void **operator=**(*nonesuch* **const**&)

Header hpx/type_support/equality.hpp

Header hpx/type_support/identity.hpp

namespace hpx

namespace util

template<typename **T**>
struct identity
#include <identity.hpp>

Public Types

typedef T **type**

Header hpx/type_support/lazy_conditional.hpp

Header hpx/type_support/lazy_enable_if.hpp

namespace hpx

namespace util

template<typename **T**>
struct lazy_enable_if<true, *T*>
#include <lazy_enable_if.hpp>

Public Types

typedef T::type **type**

Header hpx/type_support/pack.hpp

namespace hpx

namespace util

Typedefs

using index_pack = pack_c<*std*::size_t, Is...>

template<typename ...**Ts**>
struct pack
#include <pack.hpp>

Public Types

typedef pack **type**

Public Static Attributes

const *std*::size_t **size** = sizeof...(Ts)

template<typename **T**, *T*... **Vs**>
struct pack_c
#include <pack.hpp> 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

const *std*::size_t **size** = sizeof...(Vs)

Header `hpx/type_support/static.hpp`

Defines

HPX_EXPORT_STATIC_

namespace hpx

namespace util

template<typename **T**, typename **Tag** = *T*>
struct static_
#include <static.hpp>

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()

Header `hpx/type_support/unused.hpp`

Defines

HPX_UNUSED(*x*)

HPX_MAYBE_UNUSED

namespace hpx

namespace util

Variables

constexpr *unused_type* **unused** = *unused_type*()

struct unused_type
#include <unused.hpp>

Public Functions

constexpr unused_type()

constexpr unused_type(*unused_type* **const**&)

constexpr unused_type(*unused_type*&&)

template<typename **T**>
constexpr unused_type(*T* **const**&)

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*&&)

Header `hpx/type_support/unwrap_ref.hpp`

template<typename **T**>
struct unwrap_reference<*std*::reference_wrapper<*T*>>
#include <unwrap_ref.hpp>

Public Types

typedef T **type**

template<typename **T**>
struct unwrap_reference<*std*::reference_wrapper<*T*> **const**>
#include <unwrap_ref.hpp>

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
#include <unwrap_ref.hpp>

Public Types

typedef T **type**

template<typename **T**>
struct unwrap_reference<*std*::reference_wrapper<*T*>>
#include <unwrap_ref.hpp>

Public Types

typedef T **type**

template<typename **T**>
struct unwrap_reference<*std*::reference_wrapper<*T*> **const**>
#include <unwrap_ref.hpp>

Public Types

typedef T **type**

Header hpx/type_support/void_guard.hpp

namespace hpx

namespace util

template<>
struct void_guard<void>
#include <void_guard.hpp>

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.

Header hpx/util/calculate_fanout.hpp

namespace hpx

namespace util

Functions

std::size_t **calculate_fanout**(*std*::size_t *size*, *std*::size_t *local_fanout*)

Header hpx/util/get_and_reset_value.hpp

namespace hpx

namespace util

Functions

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*)

template<typename **T**>
T **get_and_reset_value**(*std*::atomic<*T*> &*value*, bool *reset*)

std::vector<*std*::int64_t> **get_and_reset_value**(*std*::vector<*std*::int64_t> &*value*, bool *reset*)

Header hpx/util/get_entry_as.hpp

namespace hpx

namespace util

Functions

template<typename DestType, typename Config, typename std::enable_if<!std::is_same<

Header hpx/util/insert_checked.hpp

namespace hpx

namespace util

Functions

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.

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**.

Header `hpx/util/ios_flags_saver.hpp`

namespace hpx

namespace util

class ios_flags_saver
#include <ios_flags_saver.hpp>

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_**

aspect_type **const a_save_**

Header hpx/util/manage_config.hpp

namespace hpx

namespace util

struct manage_config

#include <manage_config.hpp>

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_**

Header hpx/util/regex_from_pattern.hpp

namespace hpx

namespace util

Functions

std::string **regex_from_pattern**(*std*::string **const** &*pattern*, *error_code* &*ec* = *throws*)

Header hpx/util/sed_transform.hpp

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_**

Header `hpx/util/traits/await_traits.hpp`

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.

Header hpx/version.hpp

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**()

2.10 Contributing to *HPX*

HPX development happens on Github. The following sections are a collection of useful information related to *HPX* development.

2.10.1 Contributing to *HPX*

The main source of information to understand the process of how to contribute to HPX can be found in this document[241]. This is a living document that is constantly updated with relevant information.

2.10.2 *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[242] describes how that participation takes place and how to set about earning merit within the project community.

2.10.3 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. Write release notes in `docs/sphinx/releases/whats_new_$VERSION.rst`. Keep adding merged PRs and closed issues to this until just before the release is made. Use `tools/generate_pr_issue_list.sh` to generate the lists.
3. Add the new release notes to the table of contents in `docs/sphinx/releases.rst`.
4. Build the docs, and proof-read them. Update any documentation that may have changed, and correct any typos. Pay special attention to:
    - `$HPX_SOURCE/README.rst`
        - Update grant information
    - `docs/sphinx/releases/whats_new_$VERSION.rst`
    - `docs/sphinx/about_hpx/people.rst`
        - Update collaborators
        - Update grant information
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 respective packages to generate a new release to synchronize with the *HPX* release (APEX[243], libCDS[244]).
6. If there have been any commits to the release branch since the last release, create a tag from the old release branch before deleting the old release branch in the next step.
7. Unprotect the release branch in the github repository settings so that it can be deleted and recreated (tick "Allow force pushes" in the release branch settings of the repository).

[241] https://github.com/STEllAR-GROUP/hpx/blob/master/.github/CONTRIBUTING.md
[242] http://hpx.stellar-group.org/documents/governance/
[243] http://github.com/khuck/xpress-apex
[244] https://github.com/STEllAR-GROUP/libcds

8. Reset the release branch to the latest stable state on master and force push to origin/release. If you are creating a patch release, branch from the release tag for which you want to create a patch release.

   - `git checkout -b release` (or just `checkout` in case the it exists)
   - `git reset --hard stable`
   - `git push --force origin release`

9. Protect the release branch again to disable force pushes.

10. Check out the release branch.

11. Make sure `HPX_VERSION_MAJOR/MINOR/SUBMINOR` in `CMakeLists.txt` contain the correct values. Change them if needed.

12. 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).

13. Update the minimum required versions if necessary (compilers, dependencies, etc.) in `building_hpx.rst`.

14. Verify that the jenkins setup for the release branch on rostam is running and does not display any errors.

15. Repeat the following steps until satisfied with the release.

    1. Change `HPX_VERSION_TAG` in `CMakeLists.txt` to `-rcN`, where `N` is the current iteration of this step. Start with `-rc1`.

    2. Create a pre-release on GitHub using the script `tools/roll_release.sh`. This script automatically tag with the corresponding release number. The script requires that you have the STE||AR Group signing key.

    3. This step is not necessary for patch releases. Notify `hpx-users@stellar.cct.lsu.edu` and `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.

       - Use `git merge` when possible, and fall back to `git cherry-pick` when needed. For patch releases `git cherry-pick` is most likely your only choice if there have been significant unrelated changes on master since the previous release.
       - Go back to the first step when enough patches have been added.
       - If there are no more patches, continue to make the final release.

16. Update any occurrences of the latest stable release to refer to the version about to be released. For example, `quickstart.rst` contains instructions to check out the latest stable tag. Make sure that refers to the new version.

17. Add a new entry to the RPM changelog (`cmake/packaging/rpm/Changelog.txt`) with the new version number and a link to the corresponding changelog.

18. Change HPX_VERSION_TAG in CMakeLists.txt to an empty string.
19. 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.
20. Create a release on GitHub using the script tools/roll_release.sh. This script automatically tag the with the corresponding release number. The script requires that you have the STE||AR Group signing key.
21. Update the websites (stellar-group.org[245] and stellar.cct.lsu.edu[246]) with the following:
    - Download links on the downloads pages. 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).
22. Merge release branch into master.
23. 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.
24. 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
25. Announce the release on hpx-users@stellar.cct.lsu.edu, stellar@cct.lsu.edu, allcct@cct.lsu.edu, faculty@csc.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.
26. Beer and pizza.

2.10.4 Testing *HPX*

To ensure correctness of *HPX*, we ship a large variety of unit and regression tests. The tests are driven by the CTest[247] tool and are executed automatically by buildbot (see *HPX* Buildbot Website[248]) on each commit to the *HPX* Github[249] repository. In addition, it is encouraged to run the test suite manually to ensure proper operation on your target system. If a test fails for your platform, we highly recommend submitting an issue on our *HPX* Issues[250] tracker with detailed information about the target system.

[245] https://stellar-group.org
[246] https://stellar.cct.lsu.edu
[247] https://gitlab.kitware.com/cmake/community/wikis/doc/ctest/Testing-With-CTest
[248] http://rostam.cct.lsu.edu/
[249] https://github.com/STEllAR-GROUP/hpx/
[250] https://github.com/STEllAR-GROUP/hpx/issues

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[251] for further details.

Issue tracker

If you stumble over a bug or missing feature in *HPX*, please submit an issue to our *HPX* Issues[252] 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[253] page.

Continuous testing

In addition to manual testing, we run automated tests on various platforms. You can see the status of the current master head by visiting the *HPX* Buildbot Website[254]. We also run tests on all pull requests using both CircleCI[255] and a combination of CDash[256] and pycicle[257]. You can see the dashboards here: CircleCI HPX dashboard[258] and CDash HPX dashboard[259] .

2.10.5 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[260] provides a convenient alternative to quickly get all the dependencies needed to start development of *HPX*. Our testing setup on CircleCI[261] uses a docker image to run all tests.

To get started you need to install Docker[262] using whatever means is most convenient on your system. Once you have Docker[263] 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[264]. To start a container using the *HPX* build environment, run:

```
docker run --interactive --tty stellargroup/build_env:ubuntu 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[265] for more information on using Docker[266].

[251] https://www.cmake.org/cmake/help/latest/manual/ctest.1.html
[252] https://github.com/STEllAR-GROUP/hpx/issues
[253] https://stellar.cct.lsu.edu/support/
[254] http://rostam.cct.lsu.edu/
[255] https://circleci.com
[256] https://www.kitware.com/cdash/project/about.html
[257] https://github.com/biddisco/pycicle/
[258] https://circleci.com/gh/STEllAR-GROUP/hpx
[259] https://cdash.cscs.ch/index.php?project=HPX
[260] https://www.docker.com
[261] https://circleci.com
[262] https://www.docker.com
[263] https://www.docker.com
[264] https://hub.docker.com/r/stellargroup/build_env/
[265] https://docs.docker.com/
[266] https://www.docker.com

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)[267]).

2.10.6 Documentation

This documentation is built using Sphinx[268], and an automatically generated API reference using Doxygen[269] and Breathe[270].

We always welcome suggestions on how to improve our documentation, as well as pull requests with corrections and additions.

Building documentation

Please see the *documentation prerequisites* section for details on what you need in order to build the *HPX* documentation. Enable building of the documentation by setting `HPX_WITH_DOCUMENTATION=ON` during CMake[271] 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:`Component` ``.
- Indent content of directives (`.. directive::`) by three spaces.
- For C++ code samples at the end of paragraphs, use `::` and indent the code sample by 4 spaces.
  - For other languages (or if you don't want a colon at the end of the paragraph), use `.. code-block:: language` and indent by three spaces as with other directives.
- Use `.. list-table::` to wrap tables with a lot of text in cells.

[267] https://docs.docker.com/storage/bind-mounts/
[268] http://www.sphinx-doc.org
[269] https://www.doxygen.org
[270] https://breathe.readthedocs.io/en/latest
[271] 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`.

2.10.7 Module structure

This section explains the structure of an *HPX* module.

The tool create_library_skeleton.py[272] 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`

[272] 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:

- DEPRECATION_WARNINGS: Enables deprecation warnings for the module.

Optional single-value arguments are:

- COMPATIBILITY_HEADERS: Can be ON, OFF, or left out. Enables compatibility headers. Creates a variable which can be turned on or off by the user when set to ON or OFF. If left out the option is completely disabled.
- 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[273] 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[274] 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):

[273] https://www.cmake.org

[274] https://github.com/tomtom-international/cpp-dependencies

```
cpp-dependencies --dir $HPX_SOURCE/libs --graph-cycles circular_dependencies.dot
```

This will produce a `dot` file in the current directory. You can inspect this manually with a text editor. You can also convert this to an image if you have `graphviz` installed:

```
dot circular_dependencies.dot -Tsvg -o circular_dependencies.svg
```

This produces an `svg` file in the current directory which shows the circular dependencies. Note that if there are no cycles the image will be empty.

You can use `cpp-dependencies` to print the include paths between two modules.

```
cpp-dependencies --dir $HPX_SOURCE/libs --shortest <from> <to>
```

prints all possible paths from the module `<from>` to the module `<to>`. For example, as most modules depend on `config`, the following should give you a long list of paths from `algorithms` to `config`:

```
cpp-dependencies --dir $HPX_SOURCE/libs --shortest algorithms config
```

The following should report that it can't find a path between the two modules:

```
cpp-dependencies --dir $HPX_SOURCE/libs --shortest config algorithms
```

2.11 Releases

2.11.1 *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[275] - Parallel sort fails to compile with C++20
- Issue #4950[276] - Build with *HPX_WITH_PARCELPORT_ACTION_COUNTERS ON* fails
- Issue #4940[277] - Codespell report for "HPX" (on fossies.org)
- Issue #4937[278] - Allow suspension of runtime for multiple localities

[275] https://github.com/STEllAR-GROUP/hpx/issues/4971
[276] https://github.com/STEllAR-GROUP/hpx/issues/4950
[277] https://github.com/STEllAR-GROUP/hpx/issues/4940
[278] https://github.com/STEllAR-GROUP/hpx/issues/4937

Closed pull requests

- PR #4982[279] - Add page about citing HPX to documentation
- PR #4981[280] - Adding the missing include
- PR #4974[281] - Remove leftover format export hack
- PR #4972[282] - Removing use of get_temporary_buffer and return_temporary_buffer
- PR #4963[283] - Renaming files to avoid warnings from the vs build system
- PR #4951[284] - Fixing build if HPX_WITH_PARCELPORT_ACTION_COUNTERS=On
- PR #4946[285] - Allow suspension on multiple localities
- PR #4944[286] - Fix typos reported by fossies codespell report
- PR #4941[287] - Adding some explanation to README about how to cite HPX
- PR #4939[288] - Small changes

2.11.2 *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`.

279 https://github.com/STEllAR-GROUP/hpx/pull/4982
280 https://github.com/STEllAR-GROUP/hpx/pull/4981
281 https://github.com/STEllAR-GROUP/hpx/pull/4974
282 https://github.com/STEllAR-GROUP/hpx/pull/4972
283 https://github.com/STEllAR-GROUP/hpx/pull/4963
284 https://github.com/STEllAR-GROUP/hpx/pull/4951
285 https://github.com/STEllAR-GROUP/hpx/pull/4946
286 https://github.com/STEllAR-GROUP/hpx/pull/4944
287 https://github.com/STEllAR-GROUP/hpx/pull/4941
288 https://github.com/STEllAR-GROUP/hpx/pull/4939

- `result_of` and `is_callable` are now deprecated and replaced by `invoke_result` and `is_invocable` to conform to C++20.
- We continued with the modularization, making it easier for us to add the new experimental `HPX_WITH_DISTRIBUTED_RUNTIME` CMake option (see below) . An significant amount of headers have been deprecated. We adapted the namespaces and headers we could to be closer to the standard ones (*Public API*). Depending code should still compile, however warnings are now generated instructing to change the include statements accordingly.
- It is now possible to have a basic CUDA support including a helper function to get a future from a CUDA stream and target handling. They are available under the `hpx::cuda::experimental` namespace and they can be enabled with the `-DHPX_WITH_ASYNC_CUDA=ON` CMake option.
- We added a new `hpx::mpi::experimental` namespace for getting futures from an asynchronous MPI call and a new minimal MPI executor `hpx::mpi::experimental::executor`. These can be enabled with the `-DHPX_WITH_ASYNC_MPI=On` CMake option.
- A polymorphic executor has been implemented to reduce compile times as a function accepting executors can potentially be instantiated only once instead of multiple times with different executors. It accepts the function signature as a template argument. It needs to be constructed from any other executor. Please note, that the function signatures that can be scheduled using `then_execute`, `bulk_sync_execute`, `bulk_async_execute` and `bulk_then_execute` are slightly different (See the comment in PR #4514[289] 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.
- 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

289 https://github.com/STEllAR-GROUP/hpx/pull/4514

`--hpx:local` runtime option.

- We fixed task annotations for actions.
- The alias `hpx::promise` to `hpx::lcos::promise` is now deprecated. You can use `hpx::lcos::promise` directly instead. `hpx::promise` will refer to the local-only promise in the future.
- We have added a `prepare_checkpoint` API function that calculates the amount of necessary buffer space for a particular set of arguments checkpointed.
- We have added `hpx::upgrade_lock` and `hpx::upgrade_to_unique_lock`, which make `hpx::shared_mutex` (and similar) usable in more flexible ways.
- We have changed the CMake targets exposed to the user, it now includes `HPX::hpx`, `HPX::wrap_main` (`int main` as the first *HPX* thread of the application, see *Starting the HPX runtime*), `HPX::plugin`, `HPX::component`. The CMake variables `HPX_INCLUDE_DIRS` and `HPX_LIBRARIES` are deprecated and will be removed in a future release, you should now link directly to the `HPX::hpx` CMake target.
- A new example is demonstrating how to create and use a wrapping executor (`quickstart/executor_with_thread_hooks.cpp`)
- A new example is demonstrating how to disable thread stealing during the execution of parallel algorithms (`quickstart/disable_thread_stealing_executor.cpp`)
- We now require for our CMake build system configuration files to be formatted using cmake-format.
- We have removed more dependencies on various Boost libraries.
- We have added an experimental option enabling unity builds of HPX using the `-DHPX_WITH_UNITY_BUILD=On` CMake option.
- Many bug fixes.

Breaking changes

- *HPX* now requires a C++14 capable compiler. We have set the *HPX* C++ standard automatically to C++14 and if it needs to be set explicitly, it should be specified through the `CMAKE_CXX_STANDARD` setting as mandated by CMake. The `HPX_WITH_CXX*` variables are now deprecated and will be removed in the future.
- Building and using HPX is now supported only when using CMake V3.13 or later, Boost V1.64 or newer, and when compiling with clang V5, gcc V7, or VS2019, or later. Other compilers might still work but have not been tested thoroughly.
- We have added a `hpx::init_params` struct to pass parameters for *HPX* initialization e.g. the resource partitioner callback to initialize thread pools (*Using the resource partitioner*).
- The `all_to_all` algorithm is renamed to `all_gather`, and the new `all_to_all` algorithm is not compatible with the old one.
- We have moved all of the existing APIs related to resiliency into the namespace `hpx::resiliency::experimental`.

Closed issues

- Issue #4918[290] - Rename distributed_executors module
- Issue #4900[291] - Adding JOSS status badge to README
- Issue #4897[292] - Compiler warning, deprecated header used by HPX itself
- Issue #4886[293] - A future bound to an action executing on a different locality doesn't capture exception state
- Issue #4880[294] - Undefined reference to main build error when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- Issue #4877[295] - hpx_main might not able to start hpx runtime properly
- Issue #4850[296] - Issues creating templated component
- Issue #4829[297] - Spack package & HPX_WITH_GENERIC_CONTEXT_COROUTINES
- Issue #4820[298] - PAPI counters don't work
- Issue #4818[299] - HPX can't be used with IO pool turned off
- Issue #4816[300] - Build of HPX fails when find_package(Boost) is called before FetchContent_MakeAvailable(hpx)
- Issue #4813[301] - HPX MPI Future failed
- Issue #4811[302] - Remove HPX::hpx_no_wrap_main target before 1.5.0 release
- Issue #4810[303] - In hpx::for_each::invoke_projected the hpx::util::decay is misguided
- Issue #4787[304] - *transform_inclusive_scan* gives incorrect results for non-commutative operator
- Issue #4786[305] - transform_inclusive_scan tries to implicitly convert between types, instead of using the provided *conv* function
- Issue #4779[306] - HPX build error with GCC 10.1
- Issue #4766[307] - Move HPX.Compute functionality to experimental namespace
- Issue #4763[308] - License file name
- Issue #4758[309] - CMake profiling results
- Issue #4755[310] - Building HPX with support for PAPI fails

[290] https://github.com/STEllAR-GROUP/hpx/issues/4918
[291] https://github.com/STEllAR-GROUP/hpx/issues/4900
[292] https://github.com/STEllAR-GROUP/hpx/issues/4897
[293] https://github.com/STEllAR-GROUP/hpx/issues/4886
[294] https://github.com/STEllAR-GROUP/hpx/issues/4880
[295] https://github.com/STEllAR-GROUP/hpx/issues/4877
[296] https://github.com/STEllAR-GROUP/hpx/issues/4850
[297] https://github.com/STEllAR-GROUP/hpx/issues/4829
[298] https://github.com/STEllAR-GROUP/hpx/issues/4820
[299] https://github.com/STEllAR-GROUP/hpx/issues/4818
[300] https://github.com/STEllAR-GROUP/hpx/issues/4816
[301] https://github.com/STEllAR-GROUP/hpx/issues/4813
[302] https://github.com/STEllAR-GROUP/hpx/issues/4811
[303] https://github.com/STEllAR-GROUP/hpx/issues/4810
[304] https://github.com/STEllAR-GROUP/hpx/issues/4787
[305] https://github.com/STEllAR-GROUP/hpx/issues/4786
[306] https://github.com/STEllAR-GROUP/hpx/issues/4779
[307] https://github.com/STEllAR-GROUP/hpx/issues/4766
[308] https://github.com/STEllAR-GROUP/hpx/issues/4763
[309] https://github.com/STEllAR-GROUP/hpx/issues/4758
[310] https://github.com/STEllAR-GROUP/hpx/issues/4755

- Issue #4754[311] - CMake cache creation breaks when using HPX with mimalloc
- Issue #4752[312] - HPX MPI Future build failed
- Issue #4746[313] - Memory leak when using dataflow icw components
- Issue #4731[314] - Bug in stencil example, calculation of locality IDs
- Issue #4723[315] - Build fail with NETWORKING OFF
- Issue #4720[316] - Add compatibility headers for modules that had their module headers implicitly generated in 1.4.1
- Issue #4719[317] - Undeprecate some module headers
- Issue #4712[318] - Rename HPX_MPI_WITH_FUTURES option
- Issue #4709[319] - Make deprecation warnings overridable in dependent projects
- Issue #4691[320] - Suggestion to fix and enhance the thread_mapper API
- Issue #4686[321] - Fix tutorials examples
- Issue #4685[322] - HPX distributed map fails to compile
- Issue #4680[323] - Build error with HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- Issue #4679[324] - Build error for hpx w/ Apex on Summit
- Issue #4675[325] - build error with HPX_WITH_NETWORKING=OFF
- Issue #4674[326] - Error running Quickstart tests on OS X
- Issue #4662[327] - MPI initialization broken when networking off
- Issue #4652[328] - How to fix distributed action annotation
- Issue #4650[329] - thread descriptions are broken…again
- Issue #4648[330] - Thread stacksize not properly set
- Issue #4647[331] - Rename generated collective headers in modules
- Issue #4639[332] - Update deprecation warnings in compatibility headers to point to collective headers
- Issue #4628[333] - mpi parcelport totally broken

[311] https://github.com/STEllAR-GROUP/hpx/issues/4754
[312] https://github.com/STEllAR-GROUP/hpx/issues/4752
[313] https://github.com/STEllAR-GROUP/hpx/issues/4746
[314] https://github.com/STEllAR-GROUP/hpx/issues/4731
[315] https://github.com/STEllAR-GROUP/hpx/issues/4723
[316] https://github.com/STEllAR-GROUP/hpx/issues/4720
[317] https://github.com/STEllAR-GROUP/hpx/issues/4719
[318] https://github.com/STEllAR-GROUP/hpx/issues/4712
[319] https://github.com/STEllAR-GROUP/hpx/issues/4709
[320] https://github.com/STEllAR-GROUP/hpx/issues/4691
[321] https://github.com/STEllAR-GROUP/hpx/issues/4686
[322] https://github.com/STEllAR-GROUP/hpx/issues/4685
[323] https://github.com/STEllAR-GROUP/hpx/issues/4680
[324] https://github.com/STEllAR-GROUP/hpx/issues/4679
[325] https://github.com/STEllAR-GROUP/hpx/issues/4675
[326] https://github.com/STEllAR-GROUP/hpx/issues/4674
[327] https://github.com/STEllAR-GROUP/hpx/issues/4662
[328] https://github.com/STEllAR-GROUP/hpx/issues/4652
[329] https://github.com/STEllAR-GROUP/hpx/issues/4650
[330] https://github.com/STEllAR-GROUP/hpx/issues/4648
[331] https://github.com/STEllAR-GROUP/hpx/issues/4647
[332] https://github.com/STEllAR-GROUP/hpx/issues/4639
[333] https://github.com/STEllAR-GROUP/hpx/issues/4628

- Issue #4619[334] - Fully document hpx_wrap behaviour and targets
- Issue #4612[335] - Compilation issue with HPX 1.4.1 and 1.4.0
- Issue #4594[336] - Rename modules
- Issue #4578[337] - Default value for HPX_WITH_THREAD_BACKTRACE_DEPTH
- Issue #4572[338] - Thread manager should be given a runtime_configuration
- Issue #4571[339] - Add high-level documentation to new modules
- Issue #4569[340] - Annoying warning when compiling - pls suppress or fix it.
- Issue #4555[341] - HPX_HAVE_THREAD_BACKTRACE_ON_SUSPENSION compilation error
- Issue #4543[342] - Segfaults in Release builds using *sleep_for*
- Issue #4539[343] - Compilation Error when HPX_MPI_WITH_FUTURES=ON
- Issue #4537[344] - Linking issue with libhpx_initd.a
- Issue #4535[345] - API for checking if pool with a given name exists
- Issue #4523[346] - Build of PR #4311 (git tag 9955e8e) fails
- Issue #4519[347] - Documentation problem
- Issue #4513[348] - HPXConfig.cmake contains ill-formed paths when library paths use backslashes
- Issue #4507[349] - User-polling introduced by MPI futures module should be more generally usable
- Issue #4506[350] - Make sure force_linking.hpp is not included in main module header
- Issue #4501[351] - Fix compilation of PAPI tests
- Issue #4497[352] - Add modules CI checks
- Issue #4489[353] - Polymorphic executor
- Issue #4476[354] - Use CMake targets defined by FindBoost
- Issue #4473[355] - Add vcpkg installation instructions
- Issue #4470[356] - Adapt hpx::future to C++20 co_await

[334] https://github.com/STEllAR-GROUP/hpx/issues/4619
[335] https://github.com/STEllAR-GROUP/hpx/issues/4612
[336] https://github.com/STEllAR-GROUP/hpx/issues/4594
[337] https://github.com/STEllAR-GROUP/hpx/issues/4578
[338] https://github.com/STEllAR-GROUP/hpx/issues/4572
[339] https://github.com/STEllAR-GROUP/hpx/issues/4571
[340] https://github.com/STEllAR-GROUP/hpx/issues/4569
[341] https://github.com/STEllAR-GROUP/hpx/issues/4555
[342] https://github.com/STEllAR-GROUP/hpx/issues/4543
[343] https://github.com/STEllAR-GROUP/hpx/issues/4539
[344] https://github.com/STEllAR-GROUP/hpx/issues/4537
[345] https://github.com/STEllAR-GROUP/hpx/issues/4535
[346] https://github.com/STEllAR-GROUP/hpx/issues/4523
[347] https://github.com/STEllAR-GROUP/hpx/issues/4519
[348] https://github.com/STEllAR-GROUP/hpx/issues/4513
[349] https://github.com/STEllAR-GROUP/hpx/issues/4507
[350] https://github.com/STEllAR-GROUP/hpx/issues/4506
[351] https://github.com/STEllAR-GROUP/hpx/issues/4501
[352] https://github.com/STEllAR-GROUP/hpx/issues/4497
[353] https://github.com/STEllAR-GROUP/hpx/issues/4489
[354] https://github.com/STEllAR-GROUP/hpx/issues/4476
[355] https://github.com/STEllAR-GROUP/hpx/issues/4473
[356] https://github.com/STEllAR-GROUP/hpx/issues/4470

- Issue #4468[357] - Compile error on Raspberry Pi 4
- Issue #4466[358] - Compile error on Windows, current stable:
- Issue #4453[359] - Installing HPX on fedora with dnf is not adding cmake files
- Issue #4448[360] - New std::variant serialization broken
- Issue #4438[361] - Add performance counter flag is monotically increasing
- Issue #4436[362] - Build problem: same code build and works with 1.4.0 but it doesn't with 1.4.1
- Issue #4429[363] - Function descriptions not supported in distributed
- Issue #4423[364] - –hpx:ini=hpx.lock_detection=0 has no effect
- Issue #4422[365] - Add performance counter metadata
- Issue #4419[366] - Weird behavior for –hpx:print-counter-interval with large numbers
- Issue #4401[367] - Create module repository
- Issue #4400[368] - Command line options conflict related to performance counters
- Issue #4349[369] - *–hpx:use-process-mask* option throw an exception on OS X
- Issue #4345[370] - Move gh-pages branch out of hpx repo
- Issue #4323[371] - Const-correctness error in assignment operator of compute::vector
- Issue #4318[372] - ASIO breaks with C++2a concepts
- Issue #4317[373] - Application runs even if *–hpx:help* is specified
- Issue #4063[374] - Document hpxcxx compiler wrapper
- Issue #3983[375] - Implement the C++20 Synchronization Library
- Issue #3696[376] - C++11 *constexpr* support is now required
- Issue #3623[377] - Modular HPX branch and an alternative project layout
- Issue #2836[378] - The worst-case time complexity of parallel::sort seems to be O(N^2).

[357] https://github.com/STEllAR-GROUP/hpx/issues/4468
[358] https://github.com/STEllAR-GROUP/hpx/issues/4466
[359] https://github.com/STEllAR-GROUP/hpx/issues/4453
[360] https://github.com/STEllAR-GROUP/hpx/issues/4448
[361] https://github.com/STEllAR-GROUP/hpx/issues/4438
[362] https://github.com/STEllAR-GROUP/hpx/issues/4436
[363] https://github.com/STEllAR-GROUP/hpx/issues/4429
[364] https://github.com/STEllAR-GROUP/hpx/issues/4423
[365] https://github.com/STEllAR-GROUP/hpx/issues/4422
[366] https://github.com/STEllAR-GROUP/hpx/issues/4419
[367] https://github.com/STEllAR-GROUP/hpx/issues/4401
[368] https://github.com/STEllAR-GROUP/hpx/issues/4400
[369] https://github.com/STEllAR-GROUP/hpx/issues/4349
[370] https://github.com/STEllAR-GROUP/hpx/issues/4345
[371] https://github.com/STEllAR-GROUP/hpx/issues/4323
[372] https://github.com/STEllAR-GROUP/hpx/issues/4318
[373] https://github.com/STEllAR-GROUP/hpx/issues/4317
[374] https://github.com/STEllAR-GROUP/hpx/issues/4063
[375] https://github.com/STEllAR-GROUP/hpx/issues/3983
[376] https://github.com/STEllAR-GROUP/hpx/issues/3696
[377] https://github.com/STEllAR-GROUP/hpx/issues/3623
[378] https://github.com/STEllAR-GROUP/hpx/issues/2836

Closed pull requests

- PR #4936[379] - Minor documentation fixes part 2
- PR #4935[380] - Add copyright and license to joss paper file
- PR #4934[381] - Adding Semicolon in Documentation
- PR #4932[382] - Fixing compiler warnings
- PR #4931[383] - Small documentation formatting fixes
- PR #4930[384] - Documentation Distributed HPX applications localvv with local_vv
- PR #4929[385] - Add final version of the JOSS paper
- PR #4928[386] - Add HPX_NODISCARD to enable_user_polling structs
- PR #4926[387] - Rename distributed_executors module to executors_distributed
- PR #4925[388] - Making transform_reduce conforming to C++20
- PR #4923[389] - Don't acquire lock if not needed
- PR #4921[390] - Update the release notes for the release candidate 3
- PR #4920[391] - Disable libcds release
- PR #4919[392] - Make cuda event pool dynamic instead of fixed size
- PR #4917[393] - Move chrono functionality to hpx::chrono namespace
- PR #4916[394] - HPX_HAVE_DEPRECATION_WARNINGS needs to be set even when disabled
- PR #4915[395] - Moving more action related files to actions modules
- PR #4914[396] - Add alias targets with namespaces used for exporting
- PR #4912[397] - Aggregate initialize CPOs
- PR #4910[398] - Explicitly specify hwloc root on Jenkins CSCS builds
- PR #4908[399] - Fix algorithms documentation
- PR #4907[400] - Remove HPX::hpx_no_wrap_main target

[379] https://github.com/STEllAR-GROUP/hpx/pull/4936
[380] https://github.com/STEllAR-GROUP/hpx/pull/4935
[381] https://github.com/STEllAR-GROUP/hpx/pull/4934
[382] https://github.com/STEllAR-GROUP/hpx/pull/4932
[383] https://github.com/STEllAR-GROUP/hpx/pull/4931
[384] https://github.com/STEllAR-GROUP/hpx/pull/4930
[385] https://github.com/STEllAR-GROUP/hpx/pull/4929
[386] https://github.com/STEllAR-GROUP/hpx/pull/4928
[387] https://github.com/STEllAR-GROUP/hpx/pull/4926
[388] https://github.com/STEllAR-GROUP/hpx/pull/4925
[389] https://github.com/STEllAR-GROUP/hpx/pull/4923
[390] https://github.com/STEllAR-GROUP/hpx/pull/4921
[391] https://github.com/STEllAR-GROUP/hpx/pull/4920
[392] https://github.com/STEllAR-GROUP/hpx/pull/4919
[393] https://github.com/STEllAR-GROUP/hpx/pull/4917
[394] https://github.com/STEllAR-GROUP/hpx/pull/4916
[395] https://github.com/STEllAR-GROUP/hpx/pull/4915
[396] https://github.com/STEllAR-GROUP/hpx/pull/4914
[397] https://github.com/STEllAR-GROUP/hpx/pull/4912
[398] https://github.com/STEllAR-GROUP/hpx/pull/4910
[399] https://github.com/STEllAR-GROUP/hpx/pull/4908
[400] https://github.com/STEllAR-GROUP/hpx/pull/4907

- PR #4906[401] - Fixing unused variable warning
- PR #4905[402] - Adding specializations for simple for_loops
- PR #4904[403] - Update boost to 1.74.0 for the newest jenkins configs
- PR #4903[404] - Hide GITHUB_TOKEN environment variables from environment variable output
- PR #4902[405] - Cancel previous pull requests builds before starting a new one with Jenkins
- PR #4901[406] - Update public API list with updated algorithms
- PR #4899[407] - Suggested changes for HPX V1.5 release notes
- PR #4898[408] - Minor tweak to hpx::equal implementation
- PR #4896[409] - Making generate() and generate_n conforming to C++20
- PR #4895[410] - Update apex tag
- PR #4894[411] - Fix exception handling for tasks
- PR #4893[412] - Remove last use of std::result_of, removed in C++20
- PR #4892[413] - Adding replay_executor and replicate_executor
- PR #4889[414] - Restore old behaviour of not requiring linking to hpx_wrap when HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4887[415] - Making sure remotely thrown (non-hpx) exceptions are properly marshaled back to invocation site
- PR #4885[416] - Adapting hpx::find and friends to C++20
- PR #4884[417] - Adapting mismatch to C++20
- PR #4883[418] - Adapting hpx::equal to be conforming to C++20
- PR #4882[419] - Fixing exception handling for hpx::copy and adding missing tests
- PR #4881[420] - Adds different runtime exception when registering thread with the HPX runtime
- PR #4876[421] - Adding example demonstrating how to disable thread stealing during the execution of parallel algorithms
- PR #4874[422] - Adding non-policy tests to all_of, any_of, and none_of

401 https://github.com/STEllAR-GROUP/hpx/pull/4906
402 https://github.com/STEllAR-GROUP/hpx/pull/4905
403 https://github.com/STEllAR-GROUP/hpx/pull/4904
404 https://github.com/STEllAR-GROUP/hpx/pull/4903
405 https://github.com/STEllAR-GROUP/hpx/pull/4902
406 https://github.com/STEllAR-GROUP/hpx/pull/4901
407 https://github.com/STEllAR-GROUP/hpx/pull/4899
408 https://github.com/STEllAR-GROUP/hpx/pull/4898
409 https://github.com/STEllAR-GROUP/hpx/pull/4896
410 https://github.com/STEllAR-GROUP/hpx/pull/4895
411 https://github.com/STEllAR-GROUP/hpx/pull/4894
412 https://github.com/STEllAR-GROUP/hpx/pull/4893
413 https://github.com/STEllAR-GROUP/hpx/pull/4892
414 https://github.com/STEllAR-GROUP/hpx/pull/4889
415 https://github.com/STEllAR-GROUP/hpx/pull/4887
416 https://github.com/STEllAR-GROUP/hpx/pull/4885
417 https://github.com/STEllAR-GROUP/hpx/pull/4884
418 https://github.com/STEllAR-GROUP/hpx/pull/4883
419 https://github.com/STEllAR-GROUP/hpx/pull/4882
420 https://github.com/STEllAR-GROUP/hpx/pull/4881
421 https://github.com/STEllAR-GROUP/hpx/pull/4876
422 https://github.com/STEllAR-GROUP/hpx/pull/4874

- PR #4873[423] - Set CUDA compute capability on rostam Jenkins builds
- PR #4872[424] - Force partitioned vector scan tests to run serially
- PR #4870[425] - Making move conforming with C++20
- PR #4869[426] - Making destroy and destroy_n conforming to C++20
- PR #4868[427] - Fix miscellaneous header problems
- PR #4867[428] - Add CPOs for for_each
- PR #4865[429] - Adapting count and count_if to be conforming to C++20
- PR #4864[430] - Release notes 1.5.0
- PR #4863[431] - adding libcds-hpx tag to prepare for hpx1.5 release
- PR #4862[432] - Adding version specific deprecation options
- PR #4861[433] - Limiting executor improvements
- PR #4860[434] - Making fill and fill_n compatible with C++20
- PR #4859[435] - Adapting all_of, any_of, and none_of to C++20
- PR #4857[436] - Improve libCDS integration
- PR #4856[437] - Correct typos in the documentation of the hpx performance counters
- PR #4854[438] - Removing obsolete code
- PR #4853[439] - Adding test that derives component from two other components
- PR #4852[440] - Fix mpi_ring test in distributed mode by ensuring all ranks run hpx_main
- PR #4851[441] - Converting resiliency APIs to tag_invoke based CPOs
- PR #4849[442] - Enable use of future_overhead test when DISTRIBUTED_RUNTIME is OFF
- PR #4847[443] - Fixing 'error prone' constructs as reported by Codacy
- PR #4846[444] - Disable Boost.Asio concepts support
- PR #4845[445] - Fix PAPI counters

[423] https://github.com/STEllAR-GROUP/hpx/pull/4873
[424] https://github.com/STEllAR-GROUP/hpx/pull/4872
[425] https://github.com/STEllAR-GROUP/hpx/pull/4870
[426] https://github.com/STEllAR-GROUP/hpx/pull/4869
[427] https://github.com/STEllAR-GROUP/hpx/pull/4868
[428] https://github.com/STEllAR-GROUP/hpx/pull/4867
[429] https://github.com/STEllAR-GROUP/hpx/pull/4865
[430] https://github.com/STEllAR-GROUP/hpx/pull/4864
[431] https://github.com/STEllAR-GROUP/hpx/pull/4863
[432] https://github.com/STEllAR-GROUP/hpx/pull/4862
[433] https://github.com/STEllAR-GROUP/hpx/pull/4861
[434] https://github.com/STEllAR-GROUP/hpx/pull/4860
[435] https://github.com/STEllAR-GROUP/hpx/pull/4859
[436] https://github.com/STEllAR-GROUP/hpx/pull/4857
[437] https://github.com/STEllAR-GROUP/hpx/pull/4856
[438] https://github.com/STEllAR-GROUP/hpx/pull/4854
[439] https://github.com/STEllAR-GROUP/hpx/pull/4853
[440] https://github.com/STEllAR-GROUP/hpx/pull/4852
[441] https://github.com/STEllAR-GROUP/hpx/pull/4851
[442] https://github.com/STEllAR-GROUP/hpx/pull/4849
[443] https://github.com/STEllAR-GROUP/hpx/pull/4847
[444] https://github.com/STEllAR-GROUP/hpx/pull/4846
[445] https://github.com/STEllAR-GROUP/hpx/pull/4845

- PR #4843[446] - Remove dependency on various Boost headers
- PR #4841[447] - Rearrange public API headers
- PR #4840[448] - Fixing TSS problems during thread termination
- PR #4839[449] - Fix async_cuda build problems when distributed runtime is disabled
- PR #4837[450] - Restore compatibility for old (now deprecated) copy algorithms
- PR #4836[451] - Adding CPOs for hpx::reduce
- PR #4835[452] - Remove *using util::result_of* from namespace hpx
- PR #4834[453] - Fixing the calculation of the number of idle cores and the corresponding idle masks
- PR #4833[454] - Allow thread function destructors to yield
- PR #4832[455] - Fixing assertion in split_gids and memory leaks in 1d_stencil_7
- PR #4831[456] - Making sure MPI_CXX_COMPILE_FLAGS is interpreted as a sequence of options
- PR #4830[457] - Update documentation on using HPX::wrap_main
- PR #4827[458] - Update clang-newest configuration to use clang 10
- PR #4826[459] - Add Jenkins configuration for rostam
- PR #4825[460] - Move all CUDA functionality to hpx::cuda::experimental namespace
- PR #4824[461] - Add support for building master/release branches to Jenkins configuration
- PR #4821[462] - Implement customization point for hpx::copy and hpx::ranges::copy
- PR #4819[463] - Allow finding Boost components before finding HPX
- PR #4817[464] - Adding range version of stable sort
- PR #4815[465] - Fix a wrong #ifdef for IO/TIMER pools causing build errors
- PR #4814[466] - Replace hpx::function_nonser with std::function in error module
- PR #4809[467] - Foreach adapt
- PR #4808[468] - Make internal algorithms functions const

[446] https://github.com/STEllAR-GROUP/hpx/pull/4843
[447] https://github.com/STEllAR-GROUP/hpx/pull/4841
[448] https://github.com/STEllAR-GROUP/hpx/pull/4840
[449] https://github.com/STEllAR-GROUP/hpx/pull/4839
[450] https://github.com/STEllAR-GROUP/hpx/pull/4837
[451] https://github.com/STEllAR-GROUP/hpx/pull/4836
[452] https://github.com/STEllAR-GROUP/hpx/pull/4835
[453] https://github.com/STEllAR-GROUP/hpx/pull/4834
[454] https://github.com/STEllAR-GROUP/hpx/pull/4833
[455] https://github.com/STEllAR-GROUP/hpx/pull/4832
[456] https://github.com/STEllAR-GROUP/hpx/pull/4831
[457] https://github.com/STEllAR-GROUP/hpx/pull/4830
[458] https://github.com/STEllAR-GROUP/hpx/pull/4827
[459] https://github.com/STEllAR-GROUP/hpx/pull/4826
[460] https://github.com/STEllAR-GROUP/hpx/pull/4825
[461] https://github.com/STEllAR-GROUP/hpx/pull/4824
[462] https://github.com/STEllAR-GROUP/hpx/pull/4821
[463] https://github.com/STEllAR-GROUP/hpx/pull/4819
[464] https://github.com/STEllAR-GROUP/hpx/pull/4817
[465] https://github.com/STEllAR-GROUP/hpx/pull/4815
[466] https://github.com/STEllAR-GROUP/hpx/pull/4814
[467] https://github.com/STEllAR-GROUP/hpx/pull/4809
[468] https://github.com/STEllAR-GROUP/hpx/pull/4808

- PR #4807[469] - Add Jenkins configuration for running on Piz Daint
- PR #4806[470] - Update documentation links to new domain name
- PR #4805[471] - Applying changes that resolve time complexity issues in sort
- PR #4803[472] - Adding implementation of stable_sort
- PR #4802[473] - Fix datapar header paths
- PR #4801[474] - Replace boost::shared_array<T> with std::shared_ptr<T[]> if supported
- PR #4799[475] - Fixing #include paths in compatibility headers
- PR #4798[476] - Include the main module header (fixes partially #4488)
- PR #4797[477] - Change cmake targets
- PR #4794[478] - Removing 128bit integer emulation
- PR #4793[479] - Make sure global variable is handled properly
- PR #4792[480] - Replace enable_if with **HPX_CONCEPT_REQUIRES_** and add is_sentinel_for constraint
- PR #4790[481] - Move deprecation warnings from base template to template specializations for result_of etc. structs
- PR #4789[482] - Fix hangs during assertion handling and distributed runtime construction
- PR #4788[483] - Fixing inclusive transform scan algorithm to properly handle initial value
- PR #4785[484] - Fixing barrier test
- PR #4784[485] - Fixing deleter argument bindings in serialize_buffer
- PR #4783[486] - Add coveralls badge
- PR #4782[487] - Make header tests parallel again
- PR #4780[488] - Remove outdated comment about hpx::stop in documentation
- PR #4776[489] - debug print improvements
- PR #4775[490] - Checkpoint cleanup
- PR #4771[491] - Fix compilation with HPX_WITH_NETWORKING=OFF

[469] https://github.com/STEllAR-GROUP/hpx/pull/4807
[470] https://github.com/STEllAR-GROUP/hpx/pull/4806
[471] https://github.com/STEllAR-GROUP/hpx/pull/4805
[472] https://github.com/STEllAR-GROUP/hpx/pull/4803
[473] https://github.com/STEllAR-GROUP/hpx/pull/4802
[474] https://github.com/STEllAR-GROUP/hpx/pull/4801
[475] https://github.com/STEllAR-GROUP/hpx/pull/4799
[476] https://github.com/STEllAR-GROUP/hpx/pull/4798
[477] https://github.com/STEllAR-GROUP/hpx/pull/4797
[478] https://github.com/STEllAR-GROUP/hpx/pull/4794
[479] https://github.com/STEllAR-GROUP/hpx/pull/4793
[480] https://github.com/STEllAR-GROUP/hpx/pull/4792
[481] https://github.com/STEllAR-GROUP/hpx/pull/4790
[482] https://github.com/STEllAR-GROUP/hpx/pull/4789
[483] https://github.com/STEllAR-GROUP/hpx/pull/4788
[484] https://github.com/STEllAR-GROUP/hpx/pull/4785
[485] https://github.com/STEllAR-GROUP/hpx/pull/4784
[486] https://github.com/STEllAR-GROUP/hpx/pull/4783
[487] https://github.com/STEllAR-GROUP/hpx/pull/4782
[488] https://github.com/STEllAR-GROUP/hpx/pull/4780
[489] https://github.com/STEllAR-GROUP/hpx/pull/4776
[490] https://github.com/STEllAR-GROUP/hpx/pull/4775
[491] https://github.com/STEllAR-GROUP/hpx/pull/4771

- PR #4767[492] - Remove all force linking leftovers
- PR #4765[493] - Fix 1d stencil index calculation
- PR #4764[494] - Force some tests to run serially
- PR #4762[495] - Update pointees in compatibility headers
- PR #4761[496] - Fix running and building of execution module tests on CircleCI
- PR #4760[497] - Storing hpx_options in global property to speed up summary report
- PR #4759[498] - Reduce memory requirements for our main shared state
- PR #4757[499] - Fix mimalloc linking on Windows
- PR #4756[500] - Fix compilation issues
- PR #4753[501] - Re-adding API functions that were lost during merges
- PR #4751[502] - Revert "Create coverage reports and upload them to codecov.io"
- PR #4750[503] - Fixing possible race condition during termination detection
- PR #4749[504] - Deprecate result_of and friends
- PR #4748[505] - Create coverage reports and upload them to codecov.io
- PR #4747[506] - Changing #include for MPI parcelport
- PR #4745[507] - Add *is_sentinel_for* trait implementation and test
- PR #4743[508] - Fix init_globally example after runtime mode changes
- PR #4742[509] - Update SUPPORT.md
- PR #4741[510] - Fixing a warning generated for unity builds with msvc
- PR #4740[511] - Rename local_lcos and basic_execution modules
- PR #4739[512] - Undeprecate a couple of hpx/modulename.hpp headers
- PR #4738[513] - Conditionally test schedulers in thread_stacksize_current test
- PR #4734[514] - Fixing a bunch of codacy warnings

[492] https://github.com/STEllAR-GROUP/hpx/pull/4767
[493] https://github.com/STEllAR-GROUP/hpx/pull/4765
[494] https://github.com/STEllAR-GROUP/hpx/pull/4764
[495] https://github.com/STEllAR-GROUP/hpx/pull/4762
[496] https://github.com/STEllAR-GROUP/hpx/pull/4761
[497] https://github.com/STEllAR-GROUP/hpx/pull/4760
[498] https://github.com/STEllAR-GROUP/hpx/pull/4759
[499] https://github.com/STEllAR-GROUP/hpx/pull/4757
[500] https://github.com/STEllAR-GROUP/hpx/pull/4756
[501] https://github.com/STEllAR-GROUP/hpx/pull/4753
[502] https://github.com/STEllAR-GROUP/hpx/pull/4751
[503] https://github.com/STEllAR-GROUP/hpx/pull/4750
[504] https://github.com/STEllAR-GROUP/hpx/pull/4749
[505] https://github.com/STEllAR-GROUP/hpx/pull/4748
[506] https://github.com/STEllAR-GROUP/hpx/pull/4747
[507] https://github.com/STEllAR-GROUP/hpx/pull/4745
[508] https://github.com/STEllAR-GROUP/hpx/pull/4743
[509] https://github.com/STEllAR-GROUP/hpx/pull/4742
[510] https://github.com/STEllAR-GROUP/hpx/pull/4741
[511] https://github.com/STEllAR-GROUP/hpx/pull/4740
[512] https://github.com/STEllAR-GROUP/hpx/pull/4739
[513] https://github.com/STEllAR-GROUP/hpx/pull/4738
[514] https://github.com/STEllAR-GROUP/hpx/pull/4734

- PR #4733[515] - Add experimental unity build option to CMake configuration
- PR #4730[516] - Fixing compilation problems with unordered map
- PR #4729[517] - Fix APEX build
- PR #4727[518] - Fix missing runtime includes for distributed runtime
- PR #4726[519] - Add more API headers
- PR #4725[520] - Add more compatibility headers for deprecated module headers
- PR #4724[521] - Fix 4723
- PR #4721[522] - Attempt to fixing migration tests
- PR #4717[523] - Make the compatilibility headers macro conditional
- PR #4716[524] - Add hpx/runtime.hpp and hpx/distributed/runtime.hpp API headers
- PR #4714[525] - Add hpx/future.hpp header
- PR #4713[526] - Remove hpx/runtime/threads_fwd.hpp and hpx/util_fwd.hpp
- PR #4711[527] - Make module deprecation warnings overridable
- PR #4710[528] - Add compatibility headers and other fixes after module header renaming
- PR #4708[529] - Add termination handler for parallel algorithms
- PR #4707[530] - Use hpx::function_nonser instead of std::function internally
- PR #4706[531] - Move header file to module
- PR #4705[532] - Fix incorrect behaviour of cmake-format check
- PR #4704[533] - Fix resource tests
- PR #4701[534] - Fix missing includes for future::then specializations
- PR #4700[535] - Removing obsolete memory component
- PR #4699[536] - Add short descriptions to modules missing documentation
- PR #4696[537] - Rename generated modules headers

[515] https://github.com/STEllAR-GROUP/hpx/pull/4733
[516] https://github.com/STEllAR-GROUP/hpx/pull/4730
[517] https://github.com/STEllAR-GROUP/hpx/pull/4729
[518] https://github.com/STEllAR-GROUP/hpx/pull/4727
[519] https://github.com/STEllAR-GROUP/hpx/pull/4726
[520] https://github.com/STEllAR-GROUP/hpx/pull/4725
[521] https://github.com/STEllAR-GROUP/hpx/pull/4724
[522] https://github.com/STEllAR-GROUP/hpx/pull/4721
[523] https://github.com/STEllAR-GROUP/hpx/pull/4717
[524] https://github.com/STEllAR-GROUP/hpx/pull/4716
[525] https://github.com/STEllAR-GROUP/hpx/pull/4714
[526] https://github.com/STEllAR-GROUP/hpx/pull/4713
[527] https://github.com/STEllAR-GROUP/hpx/pull/4711
[528] https://github.com/STEllAR-GROUP/hpx/pull/4710
[529] https://github.com/STEllAR-GROUP/hpx/pull/4708
[530] https://github.com/STEllAR-GROUP/hpx/pull/4707
[531] https://github.com/STEllAR-GROUP/hpx/pull/4706
[532] https://github.com/STEllAR-GROUP/hpx/pull/4705
[533] https://github.com/STEllAR-GROUP/hpx/pull/4704
[534] https://github.com/STEllAR-GROUP/hpx/pull/4701
[535] https://github.com/STEllAR-GROUP/hpx/pull/4700
[536] https://github.com/STEllAR-GROUP/hpx/pull/4699
[537] https://github.com/STEllAR-GROUP/hpx/pull/4696

- PR #4693[538] - Overhauling thread_mapper for public consumption
- PR #4688[539] - Fix thread stack size handling
- PR #4687[540] - Adding all_gather and fixing all_to_all
- PR #4684[541] - Miscellaneous compilation fixes
- PR #4683[542] - Fix HPX_WITH_DYNAMIC_HPX_MAIN=OFF
- PR #4682[543] - Fix compilation of pack_traversal_rebind_container.hpp
- PR #4681[544] - Add missing hpx/execution.hpp includes for future::then
- PR #4678[545] - Typeless communicator
- PR #4677[546] - Forcing registry option to be accepted without checks.
- PR #4676[547] - Adding scatter_to/scatter_from collective operations
- PR #4673[548] - Fix PAPI counters compilation
- PR #4671[549] - Deprecate hpx::promise alias to hpx::lcos::promise
- PR #4670[550] - Explicitly instantiate get_exception
- PR #4667[551] - Add *stopValue* in *Sentinel* struct instead of *Iterator*
- PR #4666[552] - Add release build on Windows to GitHub actions
- PR #4664[553] - Creating itt_notify module.
- PR #4663[554] - Mpi fixes
- PR #4659[555] - Making sure declarations match definitions in register_locks implementation
- PR #4655[556] - Fixing task annotations for actions
- PR #4653[557] - Making sure APEX is linked into every application, if needed
- PR #4651[558] - Update get_function_annotation.hpp
- PR #4646[559] - Runtime type
- PR #4645[560] - Add a few more API headers

538 https://github.com/STEllAR-GROUP/hpx/pull/4693
539 https://github.com/STEllAR-GROUP/hpx/pull/4688
540 https://github.com/STEllAR-GROUP/hpx/pull/4687
541 https://github.com/STEllAR-GROUP/hpx/pull/4684
542 https://github.com/STEllAR-GROUP/hpx/pull/4683
543 https://github.com/STEllAR-GROUP/hpx/pull/4682
544 https://github.com/STEllAR-GROUP/hpx/pull/4681
545 https://github.com/STEllAR-GROUP/hpx/pull/4678
546 https://github.com/STEllAR-GROUP/hpx/pull/4677
547 https://github.com/STEllAR-GROUP/hpx/pull/4676
548 https://github.com/STEllAR-GROUP/hpx/pull/4673
549 https://github.com/STEllAR-GROUP/hpx/pull/4671
550 https://github.com/STEllAR-GROUP/hpx/pull/4670
551 https://github.com/STEllAR-GROUP/hpx/pull/4667
552 https://github.com/STEllAR-GROUP/hpx/pull/4666
553 https://github.com/STEllAR-GROUP/hpx/pull/4664
554 https://github.com/STEllAR-GROUP/hpx/pull/4663
555 https://github.com/STEllAR-GROUP/hpx/pull/4659
556 https://github.com/STEllAR-GROUP/hpx/pull/4655
557 https://github.com/STEllAR-GROUP/hpx/pull/4653
558 https://github.com/STEllAR-GROUP/hpx/pull/4651
559 https://github.com/STEllAR-GROUP/hpx/pull/4646
560 https://github.com/STEllAR-GROUP/hpx/pull/4645

- PR #4644[561] - Fixing support for mpirun (and similar)
- PR #4643[562] - Fixing the fix for get_idle_core_count() API
- PR #4638[563] - Remove HPX_API_EXPORT missed in previous cleanup
- PR #4636[564] - Adding C++20 barrier
- PR #4635[565] - Adding C++20 latch API
- PR #4634[566] - Adding C++20 counting semaphore API
- PR #4633[567] - Unify execution parameters customization points
- PR #4632[568] - Adding missing bulk_sync_execute wrapper to example executor
- PR #4631[569] - Updates to documentation; grammar edits.
- PR #4630[570] - Updates to documentation; moved hyperlink
- PR #4624[571] - Export set_self_ptr in thread_data.hpp instead of with forward declarations where used
- PR #4623[572] - Clean up export macros
- PR #4621[573] - Trigger an error for older boost versions on power architectures
- PR #4617[574] - Ignore user-set compatibility header options if the module does not have compatibility headers
- PR #4616[575] - Fix cmake-format warning
- PR #4615[576] - Add handler for serializing custom exceptions
- PR #4614[577] - Fix error message when HPX_IGNORE_CMAKE_BUILD_TYPE_COMPATIBILITY=OFF
- PR #4613[578] - Make partitioner constructor private
- PR #4611[579] - Making auto_chunk_size execute the given function using the given executor
- PR #4610[580] - Making sure the thread-local lock registration data is moving to the core the suspended HPX thread is resumed on
- PR #4609[581] - Adding an API function that exposes the number of idle cores
- PR #4608[582] - Fixing moodycamel namespace
- PR #4607[583] - Moving winsocket initialization to core library

[561] https://github.com/STEllAR-GROUP/hpx/pull/4644
[562] https://github.com/STEllAR-GROUP/hpx/pull/4643
[563] https://github.com/STEllAR-GROUP/hpx/pull/4638
[564] https://github.com/STEllAR-GROUP/hpx/pull/4636
[565] https://github.com/STEllAR-GROUP/hpx/pull/4635
[566] https://github.com/STEllAR-GROUP/hpx/pull/4634
[567] https://github.com/STEllAR-GROUP/hpx/pull/4633
[568] https://github.com/STEllAR-GROUP/hpx/pull/4632
[569] https://github.com/STEllAR-GROUP/hpx/pull/4631
[570] https://github.com/STEllAR-GROUP/hpx/pull/4630
[571] https://github.com/STEllAR-GROUP/hpx/pull/4624
[572] https://github.com/STEllAR-GROUP/hpx/pull/4623
[573] https://github.com/STEllAR-GROUP/hpx/pull/4621
[574] https://github.com/STEllAR-GROUP/hpx/pull/4617
[575] https://github.com/STEllAR-GROUP/hpx/pull/4616
[576] https://github.com/STEllAR-GROUP/hpx/pull/4615
[577] https://github.com/STEllAR-GROUP/hpx/pull/4614
[578] https://github.com/STEllAR-GROUP/hpx/pull/4613
[579] https://github.com/STEllAR-GROUP/hpx/pull/4611
[580] https://github.com/STEllAR-GROUP/hpx/pull/4610
[581] https://github.com/STEllAR-GROUP/hpx/pull/4609
[582] https://github.com/STEllAR-GROUP/hpx/pull/4608
[583] https://github.com/STEllAR-GROUP/hpx/pull/4607

- PR #4606[584] - Local runtime module etc.
- PR #4604[585] - Add config_registry module
- PR #4603[586] - Deal with distributed modules in their respective CMakeLists.txt
- PR #4602[587] - Small module fixes
- PR #4598[588] - Making sure current_executor and service_executor functions are linked into the core library
- PR #4597[589] - Adding broadcast_to/broadcast_from to collectives module
- PR #4596[590] - Fix performance regression in block_executor
- PR #4595[591] - Making sure main.cpp is built as a library if HPX_WITH_DYNAMIC_MAIN=OFF
- PR #4592[592] - Futures module
- PR #4591[593] - Adapting co_await support for C++20
- PR #4590[594] - Adding missing exception test for for_loop()
- PR #4587[595] - Move traits headers to hpx/modulename/traits directory
- PR #4586[596] - Remove Travis CI config
- PR #4585[597] - Update macOS test blacklist
- PR #4584[598] - Attempting to fix missing symbols in stack trace
- PR #4583[599] - Fixing bad static_cast
- PR #4582[600] - Changing download url for Windows prerequisites to circumvent bandwidth limitations
- PR #4581[601] - Adding missing using placeholder::_X
- PR #4579[602] - Move get_stack_size_name and related functions
- PR #4575[603] - Excluding unconditional definition of class backtrace from global header
- PR #4574[604] - Changing return type of hardware_concurrency() to unsigned int
- PR #4570[605] - Move tests to modules
- PR #4564[606] - Reshuffle internal targets and add HPX::hpx_no_wrap_main target

[584] https://github.com/STEllAR-GROUP/hpx/pull/4606
[585] https://github.com/STEllAR-GROUP/hpx/pull/4604
[586] https://github.com/STEllAR-GROUP/hpx/pull/4603
[587] https://github.com/STEllAR-GROUP/hpx/pull/4602
[588] https://github.com/STEllAR-GROUP/hpx/pull/4598
[589] https://github.com/STEllAR-GROUP/hpx/pull/4597
[590] https://github.com/STEllAR-GROUP/hpx/pull/4596
[591] https://github.com/STEllAR-GROUP/hpx/pull/4595
[592] https://github.com/STEllAR-GROUP/hpx/pull/4592
[593] https://github.com/STEllAR-GROUP/hpx/pull/4591
[594] https://github.com/STEllAR-GROUP/hpx/pull/4590
[595] https://github.com/STEllAR-GROUP/hpx/pull/4587
[596] https://github.com/STEllAR-GROUP/hpx/pull/4586
[597] https://github.com/STEllAR-GROUP/hpx/pull/4585
[598] https://github.com/STEllAR-GROUP/hpx/pull/4584
[599] https://github.com/STEllAR-GROUP/hpx/pull/4583
[600] https://github.com/STEllAR-GROUP/hpx/pull/4582
[601] https://github.com/STEllAR-GROUP/hpx/pull/4581
[602] https://github.com/STEllAR-GROUP/hpx/pull/4579
[603] https://github.com/STEllAR-GROUP/hpx/pull/4575
[604] https://github.com/STEllAR-GROUP/hpx/pull/4574
[605] https://github.com/STEllAR-GROUP/hpx/pull/4570
[606] https://github.com/STEllAR-GROUP/hpx/pull/4564

- PR #4563[607] - fix CMake option typo
- PR #4562[608] - Unregister lock earlier to avoid holding it while suspending
- PR #4561[609] - Adding test macros supporting custom output stream
- PR #4560[610] - Making sure hash_any::operator()() is linked into core library
- PR #4559[611] - Fixing compilation if HPX_WITH_THREAD_BACKTRACE_ON_SUSPENSION=On
- PR #4557[612] - Improve spinlock implementation to perform better in high-contention situations
- PR #4553[613] - Fix a runtime_ptr problem at shutdown when apex is enabled
- PR #4552[614] - Add configuration option for making exceptions less noisy
- PR #4551[615] - Clean up thread creation parameters
- PR #4549[616] - Test FetchContent build on GitHub actions
- PR #4548[617] - Fix stack size
- PR #4545[618] - Fix header tests
- PR #4544[619] - Fix a typo in sanitizer build
- PR #4541[620] - Add API to check if a thread pool exists
- PR #4540[621] - Making sure MPI support is enabled if MPI futures are used but networking is disabled
- PR #4538[622] - Move channel documentation examples to examples directory
- PR #4536[623] - Add generic allocator for execution policies
- PR #4534[624] - Enable compatibility headers for thread_executors module
- PR #4532[625] - Fixing broken url in README.rst
- PR #4531[626] - Update scripts
- PR #4530[627] - Make sure module API docs show up in correct order
- PR #4529[628] - Adding missing template code to module creation script
- PR #4528[629] - Make sure version module uses HPX's binary dir, not the parent's

[607] https://github.com/STEllAR-GROUP/hpx/pull/4563
[608] https://github.com/STEllAR-GROUP/hpx/pull/4562
[609] https://github.com/STEllAR-GROUP/hpx/pull/4561
[610] https://github.com/STEllAR-GROUP/hpx/pull/4560
[611] https://github.com/STEllAR-GROUP/hpx/pull/4559
[612] https://github.com/STEllAR-GROUP/hpx/pull/4557
[613] https://github.com/STEllAR-GROUP/hpx/pull/4553
[614] https://github.com/STEllAR-GROUP/hpx/pull/4552
[615] https://github.com/STEllAR-GROUP/hpx/pull/4551
[616] https://github.com/STEllAR-GROUP/hpx/pull/4549
[617] https://github.com/STEllAR-GROUP/hpx/pull/4548
[618] https://github.com/STEllAR-GROUP/hpx/pull/4545
[619] https://github.com/STEllAR-GROUP/hpx/pull/4544
[620] https://github.com/STEllAR-GROUP/hpx/pull/4541
[621] https://github.com/STEllAR-GROUP/hpx/pull/4540
[622] https://github.com/STEllAR-GROUP/hpx/pull/4538
[623] https://github.com/STEllAR-GROUP/hpx/pull/4536
[624] https://github.com/STEllAR-GROUP/hpx/pull/4534
[625] https://github.com/STEllAR-GROUP/hpx/pull/4532
[626] https://github.com/STEllAR-GROUP/hpx/pull/4531
[627] https://github.com/STEllAR-GROUP/hpx/pull/4530
[628] https://github.com/STEllAR-GROUP/hpx/pull/4529
[629] https://github.com/STEllAR-GROUP/hpx/pull/4528

- PR #4527[630] - Creating actions_base and actions module
- PR #4526[631] - Shared state for cv
- PR #4525[632] - Changing sub-name sequencing for experimental namespace
- PR #4524[633] - Add API guarantee notes to API reference documentation
- PR #4522[634] - Enable and fix deprecation warnings in execution module
- PR #4521[635] - Moves more miscellaneous files to modules
- PR #4520[636] - Skip execution customization points when executor is known
- PR #4518[637] - Module distributed lcos
- PR #4516[638] - Fix various builds
- PR #4515[639] - Replace backslashes by slashes in windows paths
- PR #4514[640] - Adding polymorphic_executor
- PR #4512[641] - Adding C++20 jthread and stop_token
- PR #4510[642] - Attempt to fix APEX linking in external packages again
- PR #4508[643] - Only test pull requests (not all branches) with GitHub actions
- PR #4505[644] - Fix duplicate linking in tests (ODR violations)
- PR #4504[645] - Fix C++ standard handling
- PR #4503[646] - Add CMakelists file check
- PR #4500[647] - Fix .clang-format version requirement comment
- PR #4499[648] - Attempting to fix hpx_init linking on macOS
- PR #4498[649] - Fix compatibility of *pool_executor*
- PR #4496[650] - Removing superfluous SPDX tags
- PR #4494[651] - Module executors
- PR #4493[652] - Pack traversal module

[630] https://github.com/STEllAR-GROUP/hpx/pull/4527
[631] https://github.com/STEllAR-GROUP/hpx/pull/4526
[632] https://github.com/STEllAR-GROUP/hpx/pull/4525
[633] https://github.com/STEllAR-GROUP/hpx/pull/4524
[634] https://github.com/STEllAR-GROUP/hpx/pull/4522
[635] https://github.com/STEllAR-GROUP/hpx/pull/4521
[636] https://github.com/STEllAR-GROUP/hpx/pull/4520
[637] https://github.com/STEllAR-GROUP/hpx/pull/4518
[638] https://github.com/STEllAR-GROUP/hpx/pull/4516
[639] https://github.com/STEllAR-GROUP/hpx/pull/4515
[640] https://github.com/STEllAR-GROUP/hpx/pull/4514
[641] https://github.com/STEllAR-GROUP/hpx/pull/4512
[642] https://github.com/STEllAR-GROUP/hpx/pull/4510
[643] https://github.com/STEllAR-GROUP/hpx/pull/4508
[644] https://github.com/STEllAR-GROUP/hpx/pull/4505
[645] https://github.com/STEllAR-GROUP/hpx/pull/4504
[646] https://github.com/STEllAR-GROUP/hpx/pull/4503
[647] https://github.com/STEllAR-GROUP/hpx/pull/4500
[648] https://github.com/STEllAR-GROUP/hpx/pull/4499
[649] https://github.com/STEllAR-GROUP/hpx/pull/4498
[650] https://github.com/STEllAR-GROUP/hpx/pull/4496
[651] https://github.com/STEllAR-GROUP/hpx/pull/4494
[652] https://github.com/STEllAR-GROUP/hpx/pull/4493

- PR #4492[653] - Update copyright year in documentation
- PR #4491[654] - Add missing current_executor header
- PR #4490[655] - Update GitHub actions configs
- PR #4487[656] - Properly dispatch exceptions thrown from hpx_main to be rethrown from hpx::init/hpx::stop
- PR #4486[657] - Fixing an initialization order problem
- PR #4485[658] - Move miscellaneous files to their rightful modules
- PR #4483[659] - Clean up imported CMake target naming
- PR #4481[660] - Add vcpkg installation instructions
- PR #4479[661] - Add hints to allow to specify MIMALLOC_ROOT
- PR #4478[662] - Async modules
- PR #4475[663] - Fix rp init changes
- PR #4474[664] - Use #pragma once in headers
- PR #4472[665] - Add more descriptive error message when using x86 coroutines on non-x86 platforms
- PR #4467[666] - Add mimalloc find cmake script
- PR #4465[667] - Add thread_executors module
- PR #4464[668] - Include module
- PR #4462[669] - Merge hpx_init and hpx_wrap into one static library
- PR #4461[670] - Making thread_data test more realistic
- PR #4460[671] - Suppress MPI warnings in version.cpp
- PR #4459[672] - Make sure pkgconfig applications link with hpx_init
- PR #4458[673] - Added example demonstrating how to create and use a wrapping executor
- PR #4457[674] - Fixing execution of thread exit functions
- PR #4456[675] - Move backtrace files to debugging module

[653] https://github.com/STEllAR-GROUP/hpx/pull/4492
[654] https://github.com/STEllAR-GROUP/hpx/pull/4491
[655] https://github.com/STEllAR-GROUP/hpx/pull/4490
[656] https://github.com/STEllAR-GROUP/hpx/pull/4487
[657] https://github.com/STEllAR-GROUP/hpx/pull/4486
[658] https://github.com/STEllAR-GROUP/hpx/pull/4485
[659] https://github.com/STEllAR-GROUP/hpx/pull/4483
[660] https://github.com/STEllAR-GROUP/hpx/pull/4481
[661] https://github.com/STEllAR-GROUP/hpx/pull/4479
[662] https://github.com/STEllAR-GROUP/hpx/pull/4478
[663] https://github.com/STEllAR-GROUP/hpx/pull/4475
[664] https://github.com/STEllAR-GROUP/hpx/pull/4474
[665] https://github.com/STEllAR-GROUP/hpx/pull/4472
[666] https://github.com/STEllAR-GROUP/hpx/pull/4467
[667] https://github.com/STEllAR-GROUP/hpx/pull/4465
[668] https://github.com/STEllAR-GROUP/hpx/pull/4464
[669] https://github.com/STEllAR-GROUP/hpx/pull/4462
[670] https://github.com/STEllAR-GROUP/hpx/pull/4461
[671] https://github.com/STEllAR-GROUP/hpx/pull/4460
[672] https://github.com/STEllAR-GROUP/hpx/pull/4459
[673] https://github.com/STEllAR-GROUP/hpx/pull/4458
[674] https://github.com/STEllAR-GROUP/hpx/pull/4457
[675] https://github.com/STEllAR-GROUP/hpx/pull/4456

- PR #4455[676] - Move deadlock_detection and maintain_queue_wait_times source files into schedulers module
- PR #4450[677] - Fixing compilation with std::filesystem enabled
- PR #4449[678] - Fixing build system to actually build variant test
- PR #4447[679] - This fixes an obsolete #include
- PR #4446[680] - Resume tasks where they were suspended
- PR #4444[681] - Minor CUDA fixes
- PR #4443[682] - Add missing tests to CircleCI config
- PR #4442[683] - Adding a tag to all auto-generated files allowing for tools to visually distinguish those
- PR #4441[684] - Adding performance counter type information
- PR #4440[685] - Fixing MSVC build
- PR #4439[686] - Link HPX::plugin and component privately in hpx_setup_target
- PR #4437[687] - Adding a test that verifies the problem can be solved using a trait specialization
- PR #4434[688] - Clean up Boost dependencies and copy string algorithms to new module
- PR #4433[689] - Fixing compilation issues (!) if MPI parcelport is enabled
- PR #4431[690] - Ignore warnings about name mangling changing
- PR #4430[691] - Add performance_counters module
- PR #4428[692] - Don't add compatibility headers to module API reference
- PR #4426[693] - Add currently failing tests on GitHub actions to blacklist
- PR #4425[694] - Clean up and correct minimum required versions
- PR #4424[695] - Making sure hpx.lock_detection=0 works as advertized
- PR #4421[696] - Making sure interval time stops underlying timer thread on termination
- PR #4417[697] - Adding serialization support for std::variant (if available) and std::tuple
- PR #4415[698] - Partially reverting changes applied by PR 4373

[676] https://github.com/STEllAR-GROUP/hpx/pull/4455
[677] https://github.com/STEllAR-GROUP/hpx/pull/4450
[678] https://github.com/STEllAR-GROUP/hpx/pull/4449
[679] https://github.com/STEllAR-GROUP/hpx/pull/4447
[680] https://github.com/STEllAR-GROUP/hpx/pull/4446
[681] https://github.com/STEllAR-GROUP/hpx/pull/4444
[682] https://github.com/STEllAR-GROUP/hpx/pull/4443
[683] https://github.com/STEllAR-GROUP/hpx/pull/4442
[684] https://github.com/STEllAR-GROUP/hpx/pull/4441
[685] https://github.com/STEllAR-GROUP/hpx/pull/4440
[686] https://github.com/STEllAR-GROUP/hpx/pull/4439
[687] https://github.com/STEllAR-GROUP/hpx/pull/4437
[688] https://github.com/STEllAR-GROUP/hpx/pull/4434
[689] https://github.com/STEllAR-GROUP/hpx/pull/4433
[690] https://github.com/STEllAR-GROUP/hpx/pull/4431
[691] https://github.com/STEllAR-GROUP/hpx/pull/4430
[692] https://github.com/STEllAR-GROUP/hpx/pull/4428
[693] https://github.com/STEllAR-GROUP/hpx/pull/4426
[694] https://github.com/STEllAR-GROUP/hpx/pull/4425
[695] https://github.com/STEllAR-GROUP/hpx/pull/4424
[696] https://github.com/STEllAR-GROUP/hpx/pull/4421
[697] https://github.com/STEllAR-GROUP/hpx/pull/4417
[698] https://github.com/STEllAR-GROUP/hpx/pull/4415

- PR #4414[699] - Added documentation for the compiler-wrapper script hpxcxx.in in creating_hpx_projects.rst
- PR #4413[700] - Merging from V1.4.1 release
- PR #4412[701] - Making sure to issue a warning if a file specified using –hpx:options-file is not found
- PR #4411[702] - Make test specific to HPX_WITH_SHARED_PRIORITY_SCHEDULER
- PR #4407[703] - Adding minimal MPI executor
- PR #4405[704] - Fix cross pool injection test, use default scheduler as falback
- PR #4404[705] - Fix a race condition and clean-up usage of scheduler mode
- PR #4399[706] - Add more threading modules
- PR #4398[707] - Add CODEOWNERS file
- PR #4395[708] - Adding a parameter to auto_chunk_size allowing to control the amount of iterations to measure
- PR #4393[709] - Use appropriate cache-line size defaults for different platforms
- PR #4391[710] - Fixing use of allocator for C++20
- PR #4390[711] - Making –hpx:help behavior consistent
- PR #4388[712] - Change the resource partitioner initialization
- PR #4387[713] - Fix roll_release.sh
- PR #4386[714] - Add warning messages for using thread binding options on macOS
- PR #4385[715] - Cuda futures
- PR #4384[716] - Make enabling dynamic hpx_main on non-Linux systems a configuration error
- PR #4383[717] - Use configure_file for HPXCacheVariables.cmake
- PR #4382[718] - Update spellchecking whitelist and fix more typos
- PR #4380[719] - Add a helper function to get a future from a cuda stream
- PR #4379[720] - Add Windows and macOS CI with GitHub actions
- PR #4378[721] - Change C++ standard handling

699 https://github.com/STEllAR-GROUP/hpx/pull/4414
700 https://github.com/STEllAR-GROUP/hpx/pull/4413
701 https://github.com/STEllAR-GROUP/hpx/pull/4412
702 https://github.com/STEllAR-GROUP/hpx/pull/4411
703 https://github.com/STEllAR-GROUP/hpx/pull/4407
704 https://github.com/STEllAR-GROUP/hpx/pull/4405
705 https://github.com/STEllAR-GROUP/hpx/pull/4404
706 https://github.com/STEllAR-GROUP/hpx/pull/4399
707 https://github.com/STEllAR-GROUP/hpx/pull/4398
708 https://github.com/STEllAR-GROUP/hpx/pull/4395
709 https://github.com/STEllAR-GROUP/hpx/pull/4393
710 https://github.com/STEllAR-GROUP/hpx/pull/4391
711 https://github.com/STEllAR-GROUP/hpx/pull/4390
712 https://github.com/STEllAR-GROUP/hpx/pull/4388
713 https://github.com/STEllAR-GROUP/hpx/pull/4387
714 https://github.com/STEllAR-GROUP/hpx/pull/4386
715 https://github.com/STEllAR-GROUP/hpx/pull/4385
716 https://github.com/STEllAR-GROUP/hpx/pull/4384
717 https://github.com/STEllAR-GROUP/hpx/pull/4383
718 https://github.com/STEllAR-GROUP/hpx/pull/4382
719 https://github.com/STEllAR-GROUP/hpx/pull/4380
720 https://github.com/STEllAR-GROUP/hpx/pull/4379
721 https://github.com/STEllAR-GROUP/hpx/pull/4378

- PR #4377[722] - Remove Python scripts
- PR #4374[723] - Adding overload for *hpx::init*/*hpx::start* for use with resource partitioner
- PR #4373[724] - Adding test that verifies for 4369 to be fixed
- PR #4372[725] - Another attempt at fixing the integral mismatch and conversion warnings
- PR #4370[726] - Doc updates quick start
- PR #4368[727] - Add a whitelist of words for weird spelling suggestions
- PR #4366[728] - Suppress or fix clang-tidy-9 warnings
- PR #4365[729] - Removing more Boost dependencies
- PR #4363[730] - Update clang-format config file for version 9
- PR #4362[731] - Fix indices typo
- PR #4361[732] - Boost cleanup
- PR #4360[733] - Move plugins
- PR #4358[734] - Doc updates; generating documentation. Will likely need heavy editing.
- PR #4356[735] - Remove some minor unused and unnecessary Boost includes
- PR #4355[736] - Fix spellcheck step in CircleCI config
- PR #4354[737] - Lightweight utility to hold a pack as members
- PR #4352[738] - Minor fixes to the C++ standard detection for MSVC
- PR #4351[739] - Move generated documentation to hpx-docs repo
- PR #4347[740] - Add cmake policy - CMP0074
- PR #4346[741] - Remove file committed by mistake
- PR #4342[742] - Remove HCC and SYCL options from CMakeLists.txt
- PR #4341[743] - Fix launch process test with APEX enabled
- PR #4340[744] - Testing Cirrus CI

[722] https://github.com/STEllAR-GROUP/hpx/pull/4377
[723] https://github.com/STEllAR-GROUP/hpx/pull/4374
[724] https://github.com/STEllAR-GROUP/hpx/pull/4373
[725] https://github.com/STEllAR-GROUP/hpx/pull/4372
[726] https://github.com/STEllAR-GROUP/hpx/pull/4370
[727] https://github.com/STEllAR-GROUP/hpx/pull/4368
[728] https://github.com/STEllAR-GROUP/hpx/pull/4366
[729] https://github.com/STEllAR-GROUP/hpx/pull/4365
[730] https://github.com/STEllAR-GROUP/hpx/pull/4363
[731] https://github.com/STEllAR-GROUP/hpx/pull/4362
[732] https://github.com/STEllAR-GROUP/hpx/pull/4361
[733] https://github.com/STEllAR-GROUP/hpx/pull/4360
[734] https://github.com/STEllAR-GROUP/hpx/pull/4358
[735] https://github.com/STEllAR-GROUP/hpx/pull/4356
[736] https://github.com/STEllAR-GROUP/hpx/pull/4355
[737] https://github.com/STEllAR-GROUP/hpx/pull/4354
[738] https://github.com/STEllAR-GROUP/hpx/pull/4352
[739] https://github.com/STEllAR-GROUP/hpx/pull/4351
[740] https://github.com/STEllAR-GROUP/hpx/pull/4347
[741] https://github.com/STEllAR-GROUP/hpx/pull/4346
[742] https://github.com/STEllAR-GROUP/hpx/pull/4342
[743] https://github.com/STEllAR-GROUP/hpx/pull/4341
[744] https://github.com/STEllAR-GROUP/hpx/pull/4340

- PR #4339[745] - Post 1.4.0 updates
- PR #4338[746] - Spelling corrections and CircleCI spell check
- PR #4333[747] - Flatten bound callables
- PR #4332[748] - This is a collection of mostly minor (cleanup) fixes
- PR #4331[749] - This adds the missing tests for async_colocated and async_continue_colocated
- PR #4330[750] - Remove HPX.Compute host default_executor
- PR #4328[751] - Generate global header for basic_execution module
- PR #4327[752] - Use INTERNAL_FLAGS option for all examples and components
- PR #4326[753] - Usage of temporary allocator in assignment operator of compute::vector
- PR #4325[754] - Use hpx::threads::get_cache_line_size in prefetching.hpp
- PR #4324[755] - Enable compatibility headers option for execution module
- PR #4316[756] - Add clang format indentppdirectives
- PR #4313[757] - Introduce index_pack alias to pack of size_t
- PR #4312[758] - Fixing compatibility header for pack.hpp
- PR #4311[759] - Dataflow annotations for APEX
- PR #4309[760] - Update launching_and_configuring_hpx_applications.rst
- PR #4306[761] - Fix schedule hint not being taken from executor
- PR #4305[762] - Implementing *hpx::functional::tag_invoke*
- PR #4304[763] - Improve pack support utilities
- PR #4303[764] - Remove errors module dependency on datastructures
- PR #4301[765] - Clean up thread executors
- PR #4294[766] - Logging revamp
- PR #4292[767] - Remove SPDX tag from Boost License file to allow for github to recognize it

745 https://github.com/STEllAR-GROUP/hpx/pull/4339
746 https://github.com/STEllAR-GROUP/hpx/pull/4338
747 https://github.com/STEllAR-GROUP/hpx/pull/4333
748 https://github.com/STEllAR-GROUP/hpx/pull/4332
749 https://github.com/STEllAR-GROUP/hpx/pull/4331
750 https://github.com/STEllAR-GROUP/hpx/pull/4330
751 https://github.com/STEllAR-GROUP/hpx/pull/4328
752 https://github.com/STEllAR-GROUP/hpx/pull/4327
753 https://github.com/STEllAR-GROUP/hpx/pull/4326
754 https://github.com/STEllAR-GROUP/hpx/pull/4325
755 https://github.com/STEllAR-GROUP/hpx/pull/4324
756 https://github.com/STEllAR-GROUP/hpx/pull/4316
757 https://github.com/STEllAR-GROUP/hpx/pull/4313
758 https://github.com/STEllAR-GROUP/hpx/pull/4312
759 https://github.com/STEllAR-GROUP/hpx/pull/4311
760 https://github.com/STEllAR-GROUP/hpx/pull/4309
761 https://github.com/STEllAR-GROUP/hpx/pull/4306
762 https://github.com/STEllAR-GROUP/hpx/pull/4305
763 https://github.com/STEllAR-GROUP/hpx/pull/4304
764 https://github.com/STEllAR-GROUP/hpx/pull/4303
765 https://github.com/STEllAR-GROUP/hpx/pull/4301
766 https://github.com/STEllAR-GROUP/hpx/pull/4294
767 https://github.com/STEllAR-GROUP/hpx/pull/4292

- PR #4291[768] - Add format support for std::tm
- PR #4290[769] - Simplify compatible tuples check
- PR #4288[770] - A lightweight take on boost::lexical_cast
- PR #4287[771] - Forking boost::lexical_cast as a new module
- PR #4277[772] - MPI_futures
- PR #4270[773] - Refactor future implementation
- PR #4265[774] - Threading module
- PR #4259[775] - Module naming base
- PR #4251[776] - Local workrequesting scheduler
- PR #4250[777] - Inline execution of scoped tasks, if possible
- PR #4247[778] - Add execution in module headers
- PR #4246[779] - Expose CMake targets officially
- PR #4239[780] - Doc updates miscellaneous (partially completed during Google Season of Docs)
- PR #4233[781] - Remove project() from modules + fix CMAKE_SOURCE_DIR issue
- PR #4231[782] - Module local lcos
- PR #4207[783] - Command line handling module
- PR #4206[784] - Runtime configuration module
- PR #4141[785] - Doc updates examples local to remote (partially completed during Google Season of Docs)
- PR #4091[786] - Split runtime into local and distributed parts
- PR #4017[787] - Require C++14

[768] https://github.com/STEllAR-GROUP/hpx/pull/4291
[769] https://github.com/STEllAR-GROUP/hpx/pull/4290
[770] https://github.com/STEllAR-GROUP/hpx/pull/4288
[771] https://github.com/STEllAR-GROUP/hpx/pull/4287
[772] https://github.com/STEllAR-GROUP/hpx/pull/4277
[773] https://github.com/STEllAR-GROUP/hpx/pull/4270
[774] https://github.com/STEllAR-GROUP/hpx/pull/4265
[775] https://github.com/STEllAR-GROUP/hpx/pull/4259
[776] https://github.com/STEllAR-GROUP/hpx/pull/4251
[777] https://github.com/STEllAR-GROUP/hpx/pull/4250
[778] https://github.com/STEllAR-GROUP/hpx/pull/4247
[779] https://github.com/STEllAR-GROUP/hpx/pull/4246
[780] https://github.com/STEllAR-GROUP/hpx/pull/4239
[781] https://github.com/STEllAR-GROUP/hpx/pull/4233
[782] https://github.com/STEllAR-GROUP/hpx/pull/4231
[783] https://github.com/STEllAR-GROUP/hpx/pull/4207
[784] https://github.com/STEllAR-GROUP/hpx/pull/4206
[785] https://github.com/STEllAR-GROUP/hpx/pull/4141
[786] https://github.com/STEllAR-GROUP/hpx/pull/4091
[787] https://github.com/STEllAR-GROUP/hpx/pull/4017

2.11.3 *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[788] - HPX 1.4.0 does not compile with gcc 10
- Issue #4336[789] - Building HPX 1.4.0 with IO Counters breaks (Windows)
- Issue #4334[790] - HPX `Debug` and `RelWithDebinfo` builds on Windows not installing `.pdb` files
- Issue #4322[791] - Undefine VT1 and VT2 after boost includes
- Issue #4314[792] - Compile error on 1.4.0
- Issue #4307[793] - `ld: error: duplicate symbol: freebsd_environ`

Closed pull requests

- PR #4376[794] - Attempt to fix some test build errors on Windows
- PR #4357[795] - Adding missing `#include`s to fix gcc V10 linker problems
- PR #4353[796] - Skip `MPI_Finalize` if `MPI_Init` is not called from HPX
- PR #4343[797] - Give a hard error if IO counters are enabled on non-Linux systems
- PR #4337[798] - Installing pdb files on Windows
- PR #4335[799] - Adding capability to buildsystem to use an installed version of HPX
- PR #4315[800] - Forcing exported symbols from composable_guard to be linked into core library
- PR #4310[801] - Remove environment handling from `exception.cpp`

[788] https://github.com/STEllAR-GROUP/hpx/issues/4320
[789] https://github.com/STEllAR-GROUP/hpx/issues/4336
[790] https://github.com/STEllAR-GROUP/hpx/issues/4334
[791] https://github.com/STEllAR-GROUP/hpx/issues/4322
[792] https://github.com/STEllAR-GROUP/hpx/issues/4314
[793] https://github.com/STEllAR-GROUP/hpx/issues/4307
[794] https://github.com/STEllAR-GROUP/hpx/pull/4376
[795] https://github.com/STEllAR-GROUP/hpx/pull/4357
[796] https://github.com/STEllAR-GROUP/hpx/pull/4353
[797] https://github.com/STEllAR-GROUP/hpx/pull/4343
[798] https://github.com/STEllAR-GROUP/hpx/pull/4337
[799] https://github.com/STEllAR-GROUP/hpx/pull/4335
[800] https://github.com/STEllAR-GROUP/hpx/pull/4315
[801] https://github.com/STEllAR-GROUP/hpx/pull/4310

2.11.4 *HPX* V1.4.0 (January 15, 2020)

General changes

- We have added the collectives `all_to_all` and `all_reduce`.
- We have added APIs for resiliency, which allows replication and replay for failed tasks. See the *documentation* for more details.
- Components can now be checkpointed.
- Performance improvements to schedulers and coroutines. A significant change is the addition of stackless coroutines. These are to be used for tasks that do not need to be suspended and can reduce overheads noticeably in applications with short tasks. A stackless coroutine can be created with the new stack size `thread_stacksize_nostack`.
- We have added an implementation of `unique_any`, which is a non-copyable version of `any`.
- The `shared_priority_queue_scheduler` has been improved. It now has lower overheads than the default scheduler in many situations. Unlike the default scheduler it fully supports NUMA scheduling hints. Enable it with the command line option `--hpx:queuing=shared-priority`. This scheduler should still be considered experimental, but its use is encouraged in real applications to help us make it production ready.
- We have added the performance counters `background-receive-duration` and `background-receive-overhead` for inspecting the time and overhead spent on receiving parcels in the background.
- Compilation time has been further improved when `HPX_WITH_NETWORKING=OFF`.
- We no longer require compiled Boost dependencies in certain configurations. This requires at least Boost 1.70, compiling on x86 with GCC 9, clang (libc++) 9, or VS2019 in C++17 mode. The dependency on Boost.Filesystem can explicitly be turned on with `HPX_FILESYSTEM_WITH_BOOST_FILESYSTEM_COMPATIBILITY=ON` (it is off by default if the standard library supports `std::filesystem`). Boost.ProgramOptions has been copied into the HPX repository. We have a compatibility layer for users who must explicitly use Boost.ProgramOptions instead of the ProgramOptions provided by HPX. To remove the dependency `HPX_PROGRAM_OPTIONS_WITH_BOOST_PROGRAM_OPTIONS_COMPATIBILITY` must be explicitly set to `OFF`. This option will be removed in a future release. We have also removed several other header-only dependencies on Boost.
- It is now possible to use the process affinity mask set by tools like `numactl` and various batch environments with the command line option `--hpx:use-process-mask`. Enabling this option implies `--hpx:ignore-batch-env`.
- It is now possible to create standalone thread pools without starting the runtime. See the `standalone_thread_pool_executor.cpp` test in the `execution` module for an example.
- Tasks annotated with `hpx::util::annotated_function` now have their correct name when using APEX to generate OTF2 files.
- Cloning of APEX was defective in previous releases (it required manual intervention to check out the correct tag or branch). This has been fixed.
- The option `HPX_WITH_MORE_THAN_64_THREADS` is now ignored and will be removed in a future release. The value is instead derived directly from `HPX_WITH_MAX_CPU_COUNT` option.
- We have deprecated compiling in C++11 mode. The next release will require a C++14 capable compiler.
- We have deprecated support for the Vc library. This option will be replaced with SIMD support from the standard library in a future release.

- We have significantly refactored our CMake setup. This is intended to be a non-breaking change and will allow for using HPX through CMake targets in the future.
- We have continued modularizing the HPX library. In the process we have rearranged many header files into module-specific directories. All moved headers have compatibility headers which forward from the old location to the new location, together with a deprecation warning. The compatibility headers will eventually be removed.
- We now enforce formatting with `clang-format` on the majority of our source files.
- We have added SPDX license tags to all files.
- Many bugfixes.

Breaking changes

- The `HPX_WITH_THREAD_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_INCLUSIVE_SCAN_COMPATIBILITY` option and the associated compatibility layer has been removed.
- The `HPX_WITH_UNWRAPPED_COMPATIBLITY` option and the associated compatibility layer has been removed.

Closed issues

- Issue #4282[802] - Build Issues with Release on Windows
- Issue #4278[803] - Build Issues with CMake 3.14.4
- Issue #4273[804] - Clients of HPX 1.4.0-rc2 with APEX ar not linked to libhpx-apex
- Issue #4269[805] - Building HPX 1.4.0-rc2 with support for APEX fails
- Issue #4263[806] - Compilation fail on latest master
- Issue #4232[807] - Configure of HPX project using CMake FetchContent fails
- Issue #4223[808] - "Re-using the main() function as the main HPX entry point" doesn't work
- Issue #4220[809] - HPX won't compile - error building `resource_partitioner`
- Issue #4215[810] - HPX 1.4.0rc1 does not link on s390x
- Issue #4204[811] - Trouble compiling HPX with Intel compiler
- Issue #4199[812] - Refactor APEX to eliminate circular dependency
- Issue #4187[813] - HPX can't build on OSX
- Issue #4185[814] - Simple debug output for development

[802] https://github.com/STEllAR-GROUP/hpx/issues/4282
[803] https://github.com/STEllAR-GROUP/hpx/issues/4278
[804] https://github.com/STEllAR-GROUP/hpx/issues/4273
[805] https://github.com/STEllAR-GROUP/hpx/issues/4269
[806] https://github.com/STEllAR-GROUP/hpx/issues/4263
[807] https://github.com/STEllAR-GROUP/hpx/issues/4232
[808] https://github.com/STEllAR-GROUP/hpx/issues/4223
[809] https://github.com/STEllAR-GROUP/hpx/issues/4220
[810] https://github.com/STEllAR-GROUP/hpx/issues/4215
[811] https://github.com/STEllAR-GROUP/hpx/issues/4204
[812] https://github.com/STEllAR-GROUP/hpx/issues/4199
[813] https://github.com/STEllAR-GROUP/hpx/issues/4187
[814] https://github.com/STEllAR-GROUP/hpx/issues/4185

- Issue #4182[815] - `@HPX_CONF_PREFIX@` is the empty string
- Issue #4169[816] - HPX won't build with APEX
- Issue #4163[817] - Add back `HPX_LIBRARIES` and `HPX_INCLUDE_DIRS`
- Issue #4161[818] - It should be possible to call `find_package(HPX)` multiple times
- Issue #4155[819] - `get_self_id()` for stackless threads returns `invalid_thread_id`
- Issue #4151[820] - build error with MPI code
- Issue #4150[821] - hpx won't build on POWER9 with clang 8
- Issue #4148[822] - `cacheline_data` delivers poor performance with C++17 compared to C++14
- Issue #4144[823] - target general in `HPX_LIBRARIES` does not exist
- Issue #4134[824] - CMake Error when `-DHPX_WITH_HPXMP=ON`
- Issue #4132[825] - parallel fill leaves elements unfilled
- Issue #4123[826] - PAPI performance counters are inaccessible
- Issue #4118[827] - `static_chunk_size` is not obeyed in scan algorithms
- Issue #4115[828] - dependency chaining error with APEX
- Issue #4107[829] - Initializing runtime without entry point function and command line arguments
- Issue #4105[830] - Bug in `hpx:bind=numa-balanced`
- Issue #4101[831] - Bound tasks
- Issue #4100[832] - Add SPDX identifier to all files
- Issue #4085[833] - `hpx_topology` library should depend on hwloc
- Issue #4067[834] - HPX fails to build on macOS
- Issue #4056[835] - Building without thread manager idle backoff fails
- Issue #4052[836] - Enforce `clang-format` style for modules
- Issue #4032[837] - Simple hello world fails to launch correctly

815 https://github.com/STEllAR-GROUP/hpx/issues/4182
816 https://github.com/STEllAR-GROUP/hpx/issues/4169
817 https://github.com/STEllAR-GROUP/hpx/issues/4163
818 https://github.com/STEllAR-GROUP/hpx/issues/4161
819 https://github.com/STEllAR-GROUP/hpx/issues/4155
820 https://github.com/STEllAR-GROUP/hpx/issues/4151
821 https://github.com/STEllAR-GROUP/hpx/issues/4150
822 https://github.com/STEllAR-GROUP/hpx/issues/4148
823 https://github.com/STEllAR-GROUP/hpx/issues/4144
824 https://github.com/STEllAR-GROUP/hpx/issues/4134
825 https://github.com/STEllAR-GROUP/hpx/issues/4132
826 https://github.com/STEllAR-GROUP/hpx/issues/4123
827 https://github.com/STEllAR-GROUP/hpx/issues/4118
828 https://github.com/STEllAR-GROUP/hpx/issues/4115
829 https://github.com/STEllAR-GROUP/hpx/issues/4107
830 https://github.com/STEllAR-GROUP/hpx/issues/4105
831 https://github.com/STEllAR-GROUP/hpx/issues/4101
832 https://github.com/STEllAR-GROUP/hpx/issues/4100
833 https://github.com/STEllAR-GROUP/hpx/issues/4085
834 https://github.com/STEllAR-GROUP/hpx/issues/4067
835 https://github.com/STEllAR-GROUP/hpx/issues/4056
836 https://github.com/STEllAR-GROUP/hpx/issues/4052
837 https://github.com/STEllAR-GROUP/hpx/issues/4032

- Issue #4030[838] - Allow threads to skip context switching
- Issue #4029[839] - Add support for mimalloc
- Issue #4005[840] - Can't link HPX when APEX enabled
- Issue #4002[841] - Missing header for algorithm module
- Issue #3989[842] - conversion from `long` to `unsigned int` requires a narrowing conversion on MSVC
- Issue #3958[843] - `/statistics/average@` perf counter can't be created
- Issue #3953[844] - CMake errors from `HPX_AddPseudoDependencies`
- Issue #3941[845] - CMake error for APEX install target
- Issue #3940[846] - Convert pseudo-doxygen function documentation into actual doxygen documentation
- Issue #3935[847] - HPX compiler match too strict?
- Issue #3929[848] - Buildbot failures on latest HPX stable
- Issue #3912[849] - I recommend publishing a version that does not depend on the boost library
- Issue #3890[850] - `hpx.ini` not working
- Issue #3883[851] - cuda compilation fails because of `-faligned-new`
- Issue #3879[852] - HPX fails to configure with `-DHPX_WITH_TESTS=OFF`
- Issue #3871[853] - `dataflow` does not support void allocators
- Issue #3867[854] - Latest HTML docs placed in wrong directory on GitHub pages
- Issue #3866[855] - Make sure all tests use `HPX_TEST*` macros and not `HPX_ASSERT`
- Issue #3857[856] - CMake all-keyword or all-plain for `target_link_libraries`
- Issue #3856[857] - `hpx_setup_target` adds rogue flags
- Issue #3850[858] - HPX fails to build on POWER8 with Clang7
- Issue #3848[859] - Remove `lva` member from `thread_init_data`
- Issue #3838[860] - `hpx::parallel::count/count_if` failing tests

838 https://github.com/STEllAR-GROUP/hpx/issues/4030
839 https://github.com/STEllAR-GROUP/hpx/issues/4029
840 https://github.com/STEllAR-GROUP/hpx/issues/4005
841 https://github.com/STEllAR-GROUP/hpx/issues/4002
842 https://github.com/STEllAR-GROUP/hpx/issues/3989
843 https://github.com/STEllAR-GROUP/hpx/issues/3958
844 https://github.com/STEllAR-GROUP/hpx/issues/3953
845 https://github.com/STEllAR-GROUP/hpx/issues/3941
846 https://github.com/STEllAR-GROUP/hpx/issues/3940
847 https://github.com/STEllAR-GROUP/hpx/issues/3935
848 https://github.com/STEllAR-GROUP/hpx/issues/3929
849 https://github.com/STEllAR-GROUP/hpx/issues/3912
850 https://github.com/STEllAR-GROUP/hpx/issues/3890
851 https://github.com/STEllAR-GROUP/hpx/issues/3883
852 https://github.com/STEllAR-GROUP/hpx/issues/3879
853 https://github.com/STEllAR-GROUP/hpx/issues/3871
854 https://github.com/STEllAR-GROUP/hpx/issues/3867
855 https://github.com/STEllAR-GROUP/hpx/issues/3866
856 https://github.com/STEllAR-GROUP/hpx/issues/3857
857 https://github.com/STEllAR-GROUP/hpx/issues/3856
858 https://github.com/STEllAR-GROUP/hpx/issues/3850
859 https://github.com/STEllAR-GROUP/hpx/issues/3848
860 https://github.com/STEllAR-GROUP/hpx/issues/3838

- Issue #3651[861] - `hpx::parallel::transform_reduce` with non const reference as lambda parameter
- Issue #3560[862] - Apex integration with HPX not working properly
- Issue #3322[863] - No warning when mixing debug/release builds

Closed pull requests

- PR #4300[864] - Checks for `MPI_Init` being called twice
- PR #4299[865] - Small CMake fixes
- PR #4298[866] - Remove extra call to annotate function that messes up traces
- PR #4296[867] - Fixing collectives locking problem
- PR #4295[868] - Do not check `LICENSE_1_0.txt` for inspect violations
- PR #4293[869] - Applying two small changes fixing carious MSVC/Windows problems
- PR #4285[870] - Delete `apex.hpp`
- PR #4276[871] - Disable doxygen generation for `hpx/debugging/print.hpp` file
- PR #4275[872] - Make sure APEX is linked to even when not explicitly referenced
- PR #4272[873] - Fix pushing of documentation
- PR #4271[874] - Updating APEX tag, don't create new task_wrapper on `operator=` of hpx_thread object
- PR #4268[875] - Testing for noexcept function specializations in C++11/14 mode
- PR #4267[876] - Fixing MSVC warning
- PR #4266[877] - Make sure macOS Travis CI fails if build step fails
- PR #4264[878] - Clean up compatibility header options
- PR #4262[879] - Cleanup modules `CMakeLists.txt`
- PR #4261[880] - Fixing HPX/APEX linking and dependencies for external projects like Phylanx
- PR #4260[881] - Fix docs compilation problems
- PR #4258[882] - Couple of minor changes

[861] https://github.com/STEllAR-GROUP/hpx/issues/3651
[862] https://github.com/STEllAR-GROUP/hpx/issues/3560
[863] https://github.com/STEllAR-GROUP/hpx/issues/3322
[864] https://github.com/STEllAR-GROUP/hpx/pull/4300
[865] https://github.com/STEllAR-GROUP/hpx/pull/4299
[866] https://github.com/STEllAR-GROUP/hpx/pull/4298
[867] https://github.com/STEllAR-GROUP/hpx/pull/4296
[868] https://github.com/STEllAR-GROUP/hpx/pull/4295
[869] https://github.com/STEllAR-GROUP/hpx/pull/4293
[870] https://github.com/STEllAR-GROUP/hpx/pull/4285
[871] https://github.com/STEllAR-GROUP/hpx/pull/4276
[872] https://github.com/STEllAR-GROUP/hpx/pull/4275
[873] https://github.com/STEllAR-GROUP/hpx/pull/4272
[874] https://github.com/STEllAR-GROUP/hpx/pull/4271
[875] https://github.com/STEllAR-GROUP/hpx/pull/4268
[876] https://github.com/STEllAR-GROUP/hpx/pull/4267
[877] https://github.com/STEllAR-GROUP/hpx/pull/4266
[878] https://github.com/STEllAR-GROUP/hpx/pull/4264
[879] https://github.com/STEllAR-GROUP/hpx/pull/4262
[880] https://github.com/STEllAR-GROUP/hpx/pull/4261
[881] https://github.com/STEllAR-GROUP/hpx/pull/4260
[882] https://github.com/STEllAR-GROUP/hpx/pull/4258

- PR #4257[883] - Fix apex annotation for async dispatch
- PR #4256[884] - Remove lambdas from assert expressions
- PR #4255[885] - Ignoring lock in `all_to_all` and `all_reduce`
- PR #4254[886] - Adding action specializations for noexcept functions
- PR #4253[887] - Move `partlit.hpp` to affinity module
- PR #4252[888] - Make mismatching build types a hard error in CMake
- PR #4249[889] - Scheduler improvement
- PR #4248[890] - update hpxmp tag to v0.3.0
- PR #4245[891] - Adding high performance channels
- PR #4244[892] - Ignore lock in ignore_while_locked_1485 test
- PR #4243[893] - Fix PAPI command line option documentation
- PR #4242[894] - Ignore lock in target_distribution_policy
- PR #4241[895] - Fix `start_stop_callbacks` test
- PR #4240[896] - Mostly fix clang CUDA compilation
- PR #4238[897] - Google Season of Docs updates to documentation; grammar edits.
- PR #4237[898] - fixing annotated task to use the name, not the desc
- PR #4236[899] - Move module print summary to modules
- PR #4235[900] - Don't use alignas in `cache_{aligned,line}_data`
- PR #4234[901] - Add basic overview sentence to all modules
- PR #4230[902] - Add OS X builds to Travis CI
- PR #4229[903] - Remove leftover queue compatibility checks
- PR #4226[904] - Fixing APEX shutdown by explicitly shutting down throttling
- PR #4225[905] - Allow `CMAKE_INSTALL_PREFIX` to be a relative path

[883] https://github.com/STEllAR-GROUP/hpx/pull/4257
[884] https://github.com/STEllAR-GROUP/hpx/pull/4256
[885] https://github.com/STEllAR-GROUP/hpx/pull/4255
[886] https://github.com/STEllAR-GROUP/hpx/pull/4254
[887] https://github.com/STEllAR-GROUP/hpx/pull/4253
[888] https://github.com/STEllAR-GROUP/hpx/pull/4252
[889] https://github.com/STEllAR-GROUP/hpx/pull/4249
[890] https://github.com/STEllAR-GROUP/hpx/pull/4248
[891] https://github.com/STEllAR-GROUP/hpx/pull/4245
[892] https://github.com/STEllAR-GROUP/hpx/pull/4244
[893] https://github.com/STEllAR-GROUP/hpx/pull/4243
[894] https://github.com/STEllAR-GROUP/hpx/pull/4242
[895] https://github.com/STEllAR-GROUP/hpx/pull/4241
[896] https://github.com/STEllAR-GROUP/hpx/pull/4240
[897] https://github.com/STEllAR-GROUP/hpx/pull/4238
[898] https://github.com/STEllAR-GROUP/hpx/pull/4237
[899] https://github.com/STEllAR-GROUP/hpx/pull/4236
[900] https://github.com/STEllAR-GROUP/hpx/pull/4235
[901] https://github.com/STEllAR-GROUP/hpx/pull/4234
[902] https://github.com/STEllAR-GROUP/hpx/pull/4230
[903] https://github.com/STEllAR-GROUP/hpx/pull/4229
[904] https://github.com/STEllAR-GROUP/hpx/pull/4226
[905] https://github.com/STEllAR-GROUP/hpx/pull/4225

- PR #4224[906] - Deprecate verbs parcelport
- PR #4222[907] - Update `register_{thread,work}` namespaces
- PR #4221[908] - Changing `HPX_GCC_VERSION` check from `70000` to `70300`
- PR #4218[909] - Google Season of Docs updates to documentation; grammar edits.
- PR #4217[910] - Google Season of Docs updates to documentation; grammar edits.
- PR #4216[911] - Fixing gcc warning on 32bit platforms (integer truncation)
- PR #4214[912] - Apex callback refactoring
- PR #4213[913] - Clean up allocator checks for dependent projects
- PR #4212[914] - Google Season of Docs updates to documentation; grammar edits.
- PR #4211[915] - Google Season of Docs updates to documentation; contributing to hpx
- PR #4210[916] - Attempting to fix Intel compilation
- PR #4209[917] - Fix CUDA 10 build
- PR #4205[918] - Making sure that differences in `CMAKE_BUILD_TYPE` are not reported on multi-configuration cmake generators
- PR #4203[919] - Deprecate Vc
- PR #4202[920] - Fix CUDA configuration
- PR #4200[921] - Making sure `hpx_wrap` is not passed on to linker on non-Linux systems
- PR #4198[922] - Fix `execution_agent.cpp` compilation with GCC 5
- PR #4197[923] - Remove deprecated options for 1.4.0 release
- PR #4196[924] - minor fixes for building on OSX Darwin
- PR #4195[925] - Use full clone on CircleCI for pushing stable tag
- PR #4193[926] - Add scheduling hints to hello_world_distributed
- PR #4192[927] - Set up CUDA in HPXConfig.cmake
- PR #4191[928] - Export allocators root variables

[906] https://github.com/STEllAR-GROUP/hpx/pull/4224
[907] https://github.com/STEllAR-GROUP/hpx/pull/4222
[908] https://github.com/STEllAR-GROUP/hpx/pull/4221
[909] https://github.com/STEllAR-GROUP/hpx/pull/4218
[910] https://github.com/STEllAR-GROUP/hpx/pull/4217
[911] https://github.com/STEllAR-GROUP/hpx/pull/4216
[912] https://github.com/STEllAR-GROUP/hpx/pull/4214
[913] https://github.com/STEllAR-GROUP/hpx/pull/4213
[914] https://github.com/STEllAR-GROUP/hpx/pull/4212
[915] https://github.com/STEllAR-GROUP/hpx/pull/4211
[916] https://github.com/STEllAR-GROUP/hpx/pull/4210
[917] https://github.com/STEllAR-GROUP/hpx/pull/4209
[918] https://github.com/STEllAR-GROUP/hpx/pull/4205
[919] https://github.com/STEllAR-GROUP/hpx/pull/4203
[920] https://github.com/STEllAR-GROUP/hpx/pull/4202
[921] https://github.com/STEllAR-GROUP/hpx/pull/4200
[922] https://github.com/STEllAR-GROUP/hpx/pull/4198
[923] https://github.com/STEllAR-GROUP/hpx/pull/4197
[924] https://github.com/STEllAR-GROUP/hpx/pull/4196
[925] https://github.com/STEllAR-GROUP/hpx/pull/4195
[926] https://github.com/STEllAR-GROUP/hpx/pull/4193
[927] https://github.com/STEllAR-GROUP/hpx/pull/4192
[928] https://github.com/STEllAR-GROUP/hpx/pull/4191

- PR #4190[929] - Don't use constexpr in thread_data with GCC <= 6
- PR #4189[930] - Only use quick_exit if available
- PR #4188[931] - Google Season of Docs updates to documentation; writing single node hpx applications
- PR #4186[932] - correct vc to cuda in cuda cmake
- PR #4184[933] - Resetting some cached variables to make sure those are re-filled
- PR #4183[934] - Fix hpxcxx configuration
- PR #4181[935] - Rename base libraries var
- PR #4180[936] - Move header left behind earlier to plugin module
- PR #4179[937] - Moving zip_iterator and transform_iterator to iterator_support module
- PR #4178[938] - Move checkpointing support to its own module
- PR #4177[939] - Small const fix to basic_execution module
- PR #4176[940] - Add back HPX_LIBRARIES and friends to HPXConfig.cmake
- PR #4175[941] - Make Vc public and add it to HPXConfig.cmake
- PR #4173[942] - Wait for runtime to be running before returning from hpx::start
- PR #4172[943] - More protection against shutdown problems in error handling scenarios.
- PR #4171[944] - Ignore lock in condition_variable::wait
- PR #4170[945] - Adding APEX dependency to MPI parcelport
- PR #4168[946] - Adding utility include
- PR #4167[947] - Add a condition to setup the external libraries
- PR #4166[948] - Add an INTERNAL_FLAGS option to link to hpx_internal_flags
- PR #4165[949] - Forward HPX_* cmake cache variables to external projects
- PR #4164[950] - Affinity and batch environment modules
- PR #4162[951] - Handle quick exit

[929] https://github.com/STEllAR-GROUP/hpx/pull/4190
[930] https://github.com/STEllAR-GROUP/hpx/pull/4189
[931] https://github.com/STEllAR-GROUP/hpx/pull/4188
[932] https://github.com/STEllAR-GROUP/hpx/pull/4186
[933] https://github.com/STEllAR-GROUP/hpx/pull/4184
[934] https://github.com/STEllAR-GROUP/hpx/pull/4183
[935] https://github.com/STEllAR-GROUP/hpx/pull/4181
[936] https://github.com/STEllAR-GROUP/hpx/pull/4180
[937] https://github.com/STEllAR-GROUP/hpx/pull/4179
[938] https://github.com/STEllAR-GROUP/hpx/pull/4178
[939] https://github.com/STEllAR-GROUP/hpx/pull/4177
[940] https://github.com/STEllAR-GROUP/hpx/pull/4176
[941] https://github.com/STEllAR-GROUP/hpx/pull/4175
[942] https://github.com/STEllAR-GROUP/hpx/pull/4173
[943] https://github.com/STEllAR-GROUP/hpx/pull/4172
[944] https://github.com/STEllAR-GROUP/hpx/pull/4171
[945] https://github.com/STEllAR-GROUP/hpx/pull/4170
[946] https://github.com/STEllAR-GROUP/hpx/pull/4168
[947] https://github.com/STEllAR-GROUP/hpx/pull/4167
[948] https://github.com/STEllAR-GROUP/hpx/pull/4166
[949] https://github.com/STEllAR-GROUP/hpx/pull/4165
[950] https://github.com/STEllAR-GROUP/hpx/pull/4164
[951] https://github.com/STEllAR-GROUP/hpx/pull/4162

- PR #4160[952] - Using `target_link_libraries` for cmake versions >= 3.12
- PR #4159[953] - Make sure `HPX_WITH_NATIVE_TLS` is forwarded to dependent projects
- PR #4158[954] - Adding allocator imported target as a dependency of allocator module
- PR #4157[955] - Add `hpx_memory` as a dependency of parcelport plugins
- PR #4156[956] - Stackless coroutines now can refer to themselves (through get_self() and friends)
- PR #4154[957] - Added CMake policy CMP0060 for HPX applications.
- PR #4153[958] - add header `iomanip` to tests and tool
- PR #4152[959] - Casting MPI tag value
- PR #4149[960] - Add back private `m_desc` member variable in program_options module
- PR #4147[961] - Resource partitioner and threadmanager modules
- PR #4146[962] - Google Season of Docs updates to documentation; creating hpx projects
- PR #4145[963] - Adding basic support for stackless threads
- PR #4143[964] - Exclude `test_client_1950` from all target
- PR #4142[965] - Add a new `thread_pool_executor`
- PR #4140[966] - Google Season of Docs updates to documentation; why hpx
- PR #4139[967] - Remove runtime includes from coroutines module
- PR #4138[968] - Forking `boost::intrusive_ptr` and adding it as `hpx::intrusive_ptr`
- PR #4137[969] - Fixing TSS destruction
- PR #4136[970] - HPX.Compute modules
- PR #4133[971] - Fix `block_executor`
- PR #4131[972] - Applying fixes based on reports from PVS Studio
- PR #4130[973] - Adding missing header to build system
- PR #4129[974] - Fixing compilation if `HPX_WITH_DATAPAR_VC` is enabled

952 https://github.com/STEllAR-GROUP/hpx/pull/4160
953 https://github.com/STEllAR-GROUP/hpx/pull/4159
954 https://github.com/STEllAR-GROUP/hpx/pull/4158
955 https://github.com/STEllAR-GROUP/hpx/pull/4157
956 https://github.com/STEllAR-GROUP/hpx/pull/4156
957 https://github.com/STEllAR-GROUP/hpx/pull/4154
958 https://github.com/STEllAR-GROUP/hpx/pull/4153
959 https://github.com/STEllAR-GROUP/hpx/pull/4152
960 https://github.com/STEllAR-GROUP/hpx/pull/4149
961 https://github.com/STEllAR-GROUP/hpx/pull/4147
962 https://github.com/STEllAR-GROUP/hpx/pull/4146
963 https://github.com/STEllAR-GROUP/hpx/pull/4145
964 https://github.com/STEllAR-GROUP/hpx/pull/4143
965 https://github.com/STEllAR-GROUP/hpx/pull/4142
966 https://github.com/STEllAR-GROUP/hpx/pull/4140
967 https://github.com/STEllAR-GROUP/hpx/pull/4139
968 https://github.com/STEllAR-GROUP/hpx/pull/4138
969 https://github.com/STEllAR-GROUP/hpx/pull/4137
970 https://github.com/STEllAR-GROUP/hpx/pull/4136
971 https://github.com/STEllAR-GROUP/hpx/pull/4133
972 https://github.com/STEllAR-GROUP/hpx/pull/4131
973 https://github.com/STEllAR-GROUP/hpx/pull/4130
974 https://github.com/STEllAR-GROUP/hpx/pull/4129

- PR #4128[975] - Renaming `moveonly_any` to `unique_any`
- PR #4126[976] - Attempt to fix `basic_any` constructor for gcc 7
- PR #4125[977] - Changing `extra_archive_data` implementation
- PR #4124[978] - Don't link to Boost.System unless required
- PR #4122[979] - Add kernel launch helper utility (+saxpy demo) and merge in octotiger changes
- PR #4121[980] - Fixing migration test if networking is disabled.
- PR #4120[981] - Google Season of Docs updates to documentation; hpx build system v1
- PR #4119[982] - Making sure `chunk_size` and `max_chunk` are actually applied to parallel algorithms if specified
- PR #4117[983] - Make CircleCI formatting check store diff
- PR #4116[984] - Fix automatically setting C++ standard
- PR #4114[985] - Module serialization
- PR #4113[986] - Module datastructures
- PR #4111[987] - Fixing performance regression introduced earlier
- PR #4110[988] - Adding missing SPDX tags
- PR #4109[989] - Overload for start without entry point/argv.
- PR #4108[990] - Making sure C++ standard is properly detected and propagated
- PR #4106[991] - use `std::round` for guaranteed rounding without errors
- PR #4104[992] - Extend `scheduler_mode` with new `work_stealing` and task assignment modes
- PR #4103[993] - Add this to lambda capture list
- PR #4102[994] - Add spdx license and check
- PR #4099[995] - Module coroutines
- PR #4098[996] - Fix append module path in module CMakeLists template
- PR #4097[997] - Function tests

975 https://github.com/STEllAR-GROUP/hpx/pull/4128
976 https://github.com/STEllAR-GROUP/hpx/pull/4126
977 https://github.com/STEllAR-GROUP/hpx/pull/4125
978 https://github.com/STEllAR-GROUP/hpx/pull/4124
979 https://github.com/STEllAR-GROUP/hpx/pull/4122
980 https://github.com/STEllAR-GROUP/hpx/pull/4121
981 https://github.com/STEllAR-GROUP/hpx/pull/4120
982 https://github.com/STEllAR-GROUP/hpx/pull/4119
983 https://github.com/STEllAR-GROUP/hpx/pull/4117
984 https://github.com/STEllAR-GROUP/hpx/pull/4116
985 https://github.com/STEllAR-GROUP/hpx/pull/4114
986 https://github.com/STEllAR-GROUP/hpx/pull/4113
987 https://github.com/STEllAR-GROUP/hpx/pull/4111
988 https://github.com/STEllAR-GROUP/hpx/pull/4110
989 https://github.com/STEllAR-GROUP/hpx/pull/4109
990 https://github.com/STEllAR-GROUP/hpx/pull/4108
991 https://github.com/STEllAR-GROUP/hpx/pull/4106
992 https://github.com/STEllAR-GROUP/hpx/pull/4104
993 https://github.com/STEllAR-GROUP/hpx/pull/4103
994 https://github.com/STEllAR-GROUP/hpx/pull/4102
995 https://github.com/STEllAR-GROUP/hpx/pull/4099
996 https://github.com/STEllAR-GROUP/hpx/pull/4098
997 https://github.com/STEllAR-GROUP/hpx/pull/4097

- PR #4096[998] - Removing return of `thread_result_type` from functions not needing them
- PR #4095[999] - Stop-gap measure until cmake overhaul is in place
- PR #4094[1000] - Deprecate `HPX_WITH_MORE_THAN_64_THREADS`
- PR #4093[1001] - Fix initialization of `global_num_tasks` in `parallel_executor`
- PR #4092[1002] - Add support for mi-malloc
- PR #4090[1003] - Execution context
- PR #4089[1004] - Make counters in coroutines optional
- PR #4087[1005] - Making `hpx::util::any` compatible with C++17
- PR #4084[1006] - Making sure destination array for `std::transform` is properly resized
- PR #4083[1007] - Adapting `thread_queue_mc` to behave even if no 128bit atomics are available
- PR #4082[1008] - Fix compilation on GCC 5
- PR #4081[1009] - Adding option allowing to force using Boost.FileSystem
- PR #4080[1010] - Updating module dependencies
- PR #4079[1011] - Add missing tests for iterator_support module
- PR #4078[1012] - Disable parcel-layer if networking is disabled
- PR #4077[1013] - Add missing include that causes build fails
- PR #4076[1014] - Enable compatibility headers for functional module
- PR #4075[1015] - Coroutines module
- PR #4073[1016] - Use `configure_file` for generated files in modules
- PR #4071[1017] - Fixing MPI detection for PMIx
- PR #4070[1018] - Fix macOS builds
- PR #4069[1019] - Moving more facilities to the collectives module
- PR #4068[1020] - Adding main HPX `#include` directory to modules

998 https://github.com/STEllAR-GROUP/hpx/pull/4096
999 https://github.com/STEllAR-GROUP/hpx/pull/4095
1000 https://github.com/STEllAR-GROUP/hpx/pull/4094
1001 https://github.com/STEllAR-GROUP/hpx/pull/4093
1002 https://github.com/STEllAR-GROUP/hpx/pull/4092
1003 https://github.com/STEllAR-GROUP/hpx/pull/4090
1004 https://github.com/STEllAR-GROUP/hpx/pull/4089
1005 https://github.com/STEllAR-GROUP/hpx/pull/4087
1006 https://github.com/STEllAR-GROUP/hpx/pull/4084
1007 https://github.com/STEllAR-GROUP/hpx/pull/4083
1008 https://github.com/STEllAR-GROUP/hpx/pull/4082
1009 https://github.com/STEllAR-GROUP/hpx/pull/4081
1010 https://github.com/STEllAR-GROUP/hpx/pull/4080
1011 https://github.com/STEllAR-GROUP/hpx/pull/4079
1012 https://github.com/STEllAR-GROUP/hpx/pull/4078
1013 https://github.com/STEllAR-GROUP/hpx/pull/4077
1014 https://github.com/STEllAR-GROUP/hpx/pull/4076
1015 https://github.com/STEllAR-GROUP/hpx/pull/4075
1016 https://github.com/STEllAR-GROUP/hpx/pull/4073
1017 https://github.com/STEllAR-GROUP/hpx/pull/4071
1018 https://github.com/STEllAR-GROUP/hpx/pull/4070
1019 https://github.com/STEllAR-GROUP/hpx/pull/4069
1020 https://github.com/STEllAR-GROUP/hpx/pull/4068

- PR #4066[1021] - Switching the use of `message(STATUS "...")` to hpx_info
- PR #4065[1022] - Move Boost.Filesystem handling to filesystem module
- PR #4064[1023] - Fix program_options test with older boost versions
- PR #4062[1024] - The `cpu_features` tool fails to compile on anything but x86 architectures
- PR #4061[1025] - Add `clang-format` checking step for modules
- PR #4060[1026] - Making sure `HPX_IDLE_BACKOFF_TIME_MAX` is always defined (even if its unused)
- PR #4059[1027] - Renaming module `hpx_parallel_executors` into `hpx_execution`
- PR #4058[1028] - Do not build networking tests when networking disabled
- PR #4057[1029] - Printing configuration summary for modules as well
- PR #4055[1030] - Google Season of Docs updates to documentation; hpx build systems
- PR #4054[1031] - Add troubleshooting section to manual
- PR #4051[1032] - Add more variations to `future_overhead` test
- PR #4050[1033] - Creating plugin module
- PR #4049[1034] - Move missing modules tests
- PR #4047[1035] - Add boost/filesystem headers to inspect deprecated headers
- PR #4045[1036] - Module functional
- PR #4043[1037] - Fix preconditions and error messages for suspension functions
- PR #4041[1038] - Pass HPX_STANDARD on to dependent projects via HPXConfig.cmake
- PR #4040[1039] - Program options module
- PR #4039[1040] - Moving non-serializable `any` (`any_nonser`) to datastructures module
- PR #4038[1041] - Adding MPark's variant (V1.4.0) to HPX
- PR #4037[1042] - Adding resiliency module
- PR #4036[1043] - Add C++17 filesystem compatibility header

[1021] https://github.com/STEllAR-GROUP/hpx/pull/4066
[1022] https://github.com/STEllAR-GROUP/hpx/pull/4065
[1023] https://github.com/STEllAR-GROUP/hpx/pull/4064
[1024] https://github.com/STEllAR-GROUP/hpx/pull/4062
[1025] https://github.com/STEllAR-GROUP/hpx/pull/4061
[1026] https://github.com/STEllAR-GROUP/hpx/pull/4060
[1027] https://github.com/STEllAR-GROUP/hpx/pull/4059
[1028] https://github.com/STEllAR-GROUP/hpx/pull/4058
[1029] https://github.com/STEllAR-GROUP/hpx/pull/4057
[1030] https://github.com/STEllAR-GROUP/hpx/pull/4055
[1031] https://github.com/STEllAR-GROUP/hpx/pull/4054
[1032] https://github.com/STEllAR-GROUP/hpx/pull/4051
[1033] https://github.com/STEllAR-GROUP/hpx/pull/4050
[1034] https://github.com/STEllAR-GROUP/hpx/pull/4049
[1035] https://github.com/STEllAR-GROUP/hpx/pull/4047
[1036] https://github.com/STEllAR-GROUP/hpx/pull/4045
[1037] https://github.com/STEllAR-GROUP/hpx/pull/4043
[1038] https://github.com/STEllAR-GROUP/hpx/pull/4041
[1039] https://github.com/STEllAR-GROUP/hpx/pull/4040
[1040] https://github.com/STEllAR-GROUP/hpx/pull/4039
[1041] https://github.com/STEllAR-GROUP/hpx/pull/4038
[1042] https://github.com/STEllAR-GROUP/hpx/pull/4037
[1043] https://github.com/STEllAR-GROUP/hpx/pull/4036

- PR #4035[1044] - Fixing support for mpirun
- PR #4028[1045] - CMake to target based directives
- PR #4027[1046] - Remove GitLab CI configuration
- PR #4026[1047] - Threading refactoring
- PR #4025[1048] - Refactoring thread queue configuration options
- PR #4024[1049] - Fix padding calculation in `cache_aligned_data.hpp`
- PR #4023[1050] - Fixing Codacy issues
- PR #4022[1051] - Make sure process mask option is passed to `affinity_data`
- PR #4021[1052] - Warn about compiling in C++11 mode
- PR #4020[1053] - Module concurrency
- PR #4019[1054] - Module topology
- PR #4018[1055] - Update deprecated header in `thread_queue_mc.hpp`
- PR #4015[1056] - Avoid overwriting artifacts
- PR #4014[1057] - Future overheads
- PR #4013[1058] - Update URL to test output conversion script
- PR #4012[1059] - Fix CUDA compilation
- PR #4011[1060] - Fixing cyclic dependencies between modules
- PR #4010[1061] - Ignore stable tag on CircleCI
- PR #4009[1062] - Check circular dependencies in a circle ci step
- PR #4008[1063] - Extend cache aligned data to handle tuple-like data
- PR #4007[1064] - Fixing migration for components that have actions returning a client
- PR #4006[1065] - Move is_value_proxy.hpp to algorithms module
- PR #4004[1066] - Shorten CTest timeout on CircleCI

[1044] https://github.com/STEllAR-GROUP/hpx/pull/4035
[1045] https://github.com/STEllAR-GROUP/hpx/pull/4028
[1046] https://github.com/STEllAR-GROUP/hpx/pull/4027
[1047] https://github.com/STEllAR-GROUP/hpx/pull/4026
[1048] https://github.com/STEllAR-GROUP/hpx/pull/4025
[1049] https://github.com/STEllAR-GROUP/hpx/pull/4024
[1050] https://github.com/STEllAR-GROUP/hpx/pull/4023
[1051] https://github.com/STEllAR-GROUP/hpx/pull/4022
[1052] https://github.com/STEllAR-GROUP/hpx/pull/4021
[1053] https://github.com/STEllAR-GROUP/hpx/pull/4020
[1054] https://github.com/STEllAR-GROUP/hpx/pull/4019
[1055] https://github.com/STEllAR-GROUP/hpx/pull/4018
[1056] https://github.com/STEllAR-GROUP/hpx/pull/4015
[1057] https://github.com/STEllAR-GROUP/hpx/pull/4014
[1058] https://github.com/STEllAR-GROUP/hpx/pull/4013
[1059] https://github.com/STEllAR-GROUP/hpx/pull/4012
[1060] https://github.com/STEllAR-GROUP/hpx/pull/4011
[1061] https://github.com/STEllAR-GROUP/hpx/pull/4010
[1062] https://github.com/STEllAR-GROUP/hpx/pull/4009
[1063] https://github.com/STEllAR-GROUP/hpx/pull/4008
[1064] https://github.com/STEllAR-GROUP/hpx/pull/4007
[1065] https://github.com/STEllAR-GROUP/hpx/pull/4006
[1066] https://github.com/STEllAR-GROUP/hpx/pull/4004

- PR #4003[1067] - Refactoring to remove (internal) dependencies
- PR #4001[1068] - Exclude tests from all target
- PR #4000[1069] - Module errors
- PR #3999[1070] - Enable support for compatibility headers for logging module
- PR #3998[1071] - Add process thread binding option
- PR #3997[1072] - Export handle_assert function
- PR #3996[1073] - Attempt to solve issue where `-latomic` does not support 128bit atomics
- PR #3993[1074] - Make sure `__LINE__` is an unsigned
- PR #3991[1075] - Fix dependencies and flags for header tests
- PR #3990[1076] - Documentation tags fixes
- PR #3988[1077] - Adding missing solution folder for format module test
- PR #3987[1078] - Move runtime-dependent functions out of command line handling
- PR #3986[1079] - Fix CMake configuration with PAPI on
- PR #3985[1080] - Module timing
- PR #3984[1081] - Fix default behaviour of paths in `add_hpx_component`
- PR #3982[1082] - Parallel executors module
- PR #3981[1083] - Segmented algorithms module
- PR #3980[1084] - Module logging
- PR #3979[1085] - Module util
- PR #3978[1086] - Fix `clang-tidy` step on CircleCI
- PR #3977[1087] - Fixing solution folders for moved components
- PR #3976[1088] - Module format
- PR #3975[1089] - Enable deprecation warnings on CircleCI

[1067] https://github.com/STEllAR-GROUP/hpx/pull/4003
[1068] https://github.com/STEllAR-GROUP/hpx/pull/4001
[1069] https://github.com/STEllAR-GROUP/hpx/pull/4000
[1070] https://github.com/STEllAR-GROUP/hpx/pull/3999
[1071] https://github.com/STEllAR-GROUP/hpx/pull/3998
[1072] https://github.com/STEllAR-GROUP/hpx/pull/3997
[1073] https://github.com/STEllAR-GROUP/hpx/pull/3996
[1074] https://github.com/STEllAR-GROUP/hpx/pull/3993
[1075] https://github.com/STEllAR-GROUP/hpx/pull/3991
[1076] https://github.com/STEllAR-GROUP/hpx/pull/3990
[1077] https://github.com/STEllAR-GROUP/hpx/pull/3988
[1078] https://github.com/STEllAR-GROUP/hpx/pull/3987
[1079] https://github.com/STEllAR-GROUP/hpx/pull/3986
[1080] https://github.com/STEllAR-GROUP/hpx/pull/3985
[1081] https://github.com/STEllAR-GROUP/hpx/pull/3984
[1082] https://github.com/STEllAR-GROUP/hpx/pull/3982
[1083] https://github.com/STEllAR-GROUP/hpx/pull/3981
[1084] https://github.com/STEllAR-GROUP/hpx/pull/3980
[1085] https://github.com/STEllAR-GROUP/hpx/pull/3979
[1086] https://github.com/STEllAR-GROUP/hpx/pull/3978
[1087] https://github.com/STEllAR-GROUP/hpx/pull/3977
[1088] https://github.com/STEllAR-GROUP/hpx/pull/3976
[1089] https://github.com/STEllAR-GROUP/hpx/pull/3975

- PR #3974[1090] - Fix typos in documentation
- PR #3973[1091] - Fix compilation with GCC 9
- PR #3972[1092] - Add condition to clone apex + use of new cmake var APEX_ROOT
- PR #3971[1093] - Add testing module
- PR #3968[1094] - Remove unneeded file in hardware module
- PR #3967[1095] - Remove leftover PIC settings from main CMakeLists.txt
- PR #3966[1096] - Add missing export option in `add_hpx_module`
- PR #3965[1097] - Change `current_function_helper` back to non-constexpr
- PR #3964[1098] - Fixing merge problems
- PR #3962[1099] - Add a trait for `std::array` for unwrapping
- PR #3961[1100] - Making `hpx::util::tuple<Ts...>` and `std::tuple<Ts...>` convertible
- PR #3960[1101] - fix compilation with CUDA 10 and GCC 6
- PR #3959[1102] - Fix C++11 incompatibility
- PR #3957[1103] - Algorithms module
- PR #3956[1104] - [`HPX_AddModule`] Fix lower name var to upper
- PR #3955[1105] - Fix CMake configuration with examples off and tests on
- PR #3954[1106] - Move components to separate subdirectory in root of repository
- PR #3952[1107] - Update `papi.cpp`
- PR #3951[1108] - Exclude modules header tests from all target
- PR #3950[1109] - Adding `all_reduce` facility to collectives module
- PR #3949[1110] - This adds a configuration file that will cause for stale issues to be automatically closed
- PR #3948[1111] - Fixing ALPS environment
- PR #3947[1112] - Add major compiler version check for building hpx as a binary package

[1090] https://github.com/STEllAR-GROUP/hpx/pull/3974
[1091] https://github.com/STEllAR-GROUP/hpx/pull/3973
[1092] https://github.com/STEllAR-GROUP/hpx/pull/3972
[1093] https://github.com/STEllAR-GROUP/hpx/pull/3971
[1094] https://github.com/STEllAR-GROUP/hpx/pull/3968
[1095] https://github.com/STEllAR-GROUP/hpx/pull/3967
[1096] https://github.com/STEllAR-GROUP/hpx/pull/3966
[1097] https://github.com/STEllAR-GROUP/hpx/pull/3965
[1098] https://github.com/STEllAR-GROUP/hpx/pull/3964
[1099] https://github.com/STEllAR-GROUP/hpx/pull/3962
[1100] https://github.com/STEllAR-GROUP/hpx/pull/3961
[1101] https://github.com/STEllAR-GROUP/hpx/pull/3960
[1102] https://github.com/STEllAR-GROUP/hpx/pull/3959
[1103] https://github.com/STEllAR-GROUP/hpx/pull/3957
[1104] https://github.com/STEllAR-GROUP/hpx/pull/3956
[1105] https://github.com/STEllAR-GROUP/hpx/pull/3955
[1106] https://github.com/STEllAR-GROUP/hpx/pull/3954
[1107] https://github.com/STEllAR-GROUP/hpx/pull/3952
[1108] https://github.com/STEllAR-GROUP/hpx/pull/3951
[1109] https://github.com/STEllAR-GROUP/hpx/pull/3950
[1110] https://github.com/STEllAR-GROUP/hpx/pull/3949
[1111] https://github.com/STEllAR-GROUP/hpx/pull/3948
[1112] https://github.com/STEllAR-GROUP/hpx/pull/3947

- PR #3946[1113] - [Modules] Move the location of the generated headers
- PR #3945[1114] - Simplify tests and examples cmake
- PR #3943[1115] - Remove example module
- PR #3942[1116] - Add `NOEXPORT` option to `add_hpx_{component,library}`
- PR #3938[1117] - Use https for CDash submissions
- PR #3937[1118] - Add `HPX_WITH_BUILD_BINARY_PACKAGE` to the compiler check (refs #3935)
- PR #3936[1119] - Fixing installation of binaries on windows
- PR #3934[1120] - Add set function for `sliding_semaphore` `max_difference`
- PR #3933[1121] - Remove `cudadevrt` from compile/link flags as it breaks downstream projects
- PR #3932[1122] - Fixing 3929
- PR #3931[1123] - Adding `all_to_all`
- PR #3930[1124] - Add test demonstrating the use of broadcast with component actions
- PR #3928[1125] - fixed number of tasks and number of threads for heterogeneous slurm environments
- PR #3927[1126] - Moving Cache module's tests into separate solution folder
- PR #3926[1127] - Move unit tests to cache module
- PR #3925[1128] - Move version check to config module
- PR #3924[1129] - Add schedule hint executor parameters
- PR #3923[1130] - Allow aligning objects bigger than the cache line size
- PR #3922[1131] - Add Windows builds with Travis CI
- PR #3921[1132] - Add ccls cache directory to gitignore
- PR #3920[1133] - Fix `git_external` fetching of tags
- PR #3905[1134] - Correct rostambod url. Fix typo in doc
- PR #3904[1135] - Fix bug in context_base.hpp

[1113] https://github.com/STEllAR-GROUP/hpx/pull/3946
[1114] https://github.com/STEllAR-GROUP/hpx/pull/3945
[1115] https://github.com/STEllAR-GROUP/hpx/pull/3943
[1116] https://github.com/STEllAR-GROUP/hpx/pull/3942
[1117] https://github.com/STEllAR-GROUP/hpx/pull/3938
[1118] https://github.com/STEllAR-GROUP/hpx/pull/3937
[1119] https://github.com/STEllAR-GROUP/hpx/pull/3936
[1120] https://github.com/STEllAR-GROUP/hpx/pull/3934
[1121] https://github.com/STEllAR-GROUP/hpx/pull/3933
[1122] https://github.com/STEllAR-GROUP/hpx/pull/3932
[1123] https://github.com/STEllAR-GROUP/hpx/pull/3931
[1124] https://github.com/STEllAR-GROUP/hpx/pull/3930
[1125] https://github.com/STEllAR-GROUP/hpx/pull/3928
[1126] https://github.com/STEllAR-GROUP/hpx/pull/3927
[1127] https://github.com/STEllAR-GROUP/hpx/pull/3926
[1128] https://github.com/STEllAR-GROUP/hpx/pull/3925
[1129] https://github.com/STEllAR-GROUP/hpx/pull/3924
[1130] https://github.com/STEllAR-GROUP/hpx/pull/3923
[1131] https://github.com/STEllAR-GROUP/hpx/pull/3922
[1132] https://github.com/STEllAR-GROUP/hpx/pull/3921
[1133] https://github.com/STEllAR-GROUP/hpx/pull/3920
[1134] https://github.com/STEllAR-GROUP/hpx/pull/3905
[1135] https://github.com/STEllAR-GROUP/hpx/pull/3904

- PR #3903[1136] - Adding new performance counters
- PR #3902[1137] - Add `add_hpx_module` function
- PR #3901[1138] - Factoring out container remapping into a separate trait
- PR #3900[1139] - Making sure errors during command line processing are properly reported and will not cause assertions
- PR #3899[1140] - Remove old compatibility bases from `make_action`
- PR #3898[1141] - Make parameter size be of type `size_t`
- PR #3897[1142] - Making sure all tests are disabled if `HPX_WITH_TESTS=OFF`
- PR #3895[1143] - Add documentation for annotated_function
- PR #3894[1144] - Working around VS2019 problem with `make_action`
- PR #3892[1145] - Avoid MSVC compatibility warning in internal allocator
- PR #3891[1146] - Removal of the default intel config include
- PR #3888[1147] - Fix `async_customization` dataflow example and Clarify what's being tested
- PR #3887[1148] - Add Doxygen documentation
- PR #3882[1149] - Minor docs fixes
- PR #3880[1150] - Updating APEX version tag
- PR #3878[1151] - Making sure symbols are properly exported from modules (needed for Windows/MacOS)
- PR #3877[1152] - Documentation
- PR #3876[1153] - Module hardware
- PR #3875[1154] - Converted typedefs in actions submodule to using directives
- PR #3874[1155] - Allow one to suppress target keywords in `hpx_setup_target` for backwards compatibility
- PR #3873[1156] - Add scripts to create releases and generate lists of PRs and issues
- PR #3872[1157] - Fix latest HTML docs location
- PR #3870[1158] - Module cache

1136 https://github.com/STEllAR-GROUP/hpx/pull/3903
1137 https://github.com/STEllAR-GROUP/hpx/pull/3902
1138 https://github.com/STEllAR-GROUP/hpx/pull/3901
1139 https://github.com/STEllAR-GROUP/hpx/pull/3900
1140 https://github.com/STEllAR-GROUP/hpx/pull/3899
1141 https://github.com/STEllAR-GROUP/hpx/pull/3898
1142 https://github.com/STEllAR-GROUP/hpx/pull/3897
1143 https://github.com/STEllAR-GROUP/hpx/pull/3895
1144 https://github.com/STEllAR-GROUP/hpx/pull/3894
1145 https://github.com/STEllAR-GROUP/hpx/pull/3892
1146 https://github.com/STEllAR-GROUP/hpx/pull/3891
1147 https://github.com/STEllAR-GROUP/hpx/pull/3888
1148 https://github.com/STEllAR-GROUP/hpx/pull/3887
1149 https://github.com/STEllAR-GROUP/hpx/pull/3882
1150 https://github.com/STEllAR-GROUP/hpx/pull/3880
1151 https://github.com/STEllAR-GROUP/hpx/pull/3878
1152 https://github.com/STEllAR-GROUP/hpx/pull/3877
1153 https://github.com/STEllAR-GROUP/hpx/pull/3876
1154 https://github.com/STEllAR-GROUP/hpx/pull/3875
1155 https://github.com/STEllAR-GROUP/hpx/pull/3874
1156 https://github.com/STEllAR-GROUP/hpx/pull/3873
1157 https://github.com/STEllAR-GROUP/hpx/pull/3872
1158 https://github.com/STEllAR-GROUP/hpx/pull/3870

- PR #3869[1159] - Post 1.3.0 version bumps
- PR #3868[1160] - Replace the macro `HPX_ASSERT` by `HPX_TEST` in tests
- PR #3845[1161] - Assertion module
- PR #3839[1162] - Make tuple serialization non-intrusive
- PR #3832[1163] - Config module
- PR #3799[1164] - Remove compat namespace and its contents
- PR #3701[1165] - MoodyCamel lockfree
- PR #3496[1166] - Disabling MPI's (deprecated) C++ interface
- PR #3192[1167] - Move type info into `hpx::debug` namespace and add print helper functions
- PR #3159[1168] - Support Checkpointing Components

2.11.5 *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.

[1159] https://github.com/STEllAR-GROUP/hpx/pull/3869
[1160] https://github.com/STEllAR-GROUP/hpx/pull/3868
[1161] https://github.com/STEllAR-GROUP/hpx/pull/3845
[1162] https://github.com/STEllAR-GROUP/hpx/pull/3839
[1163] https://github.com/STEllAR-GROUP/hpx/pull/3832
[1164] https://github.com/STEllAR-GROUP/hpx/pull/3799
[1165] https://github.com/STEllAR-GROUP/hpx/pull/3701
[1166] https://github.com/STEllAR-GROUP/hpx/pull/3496
[1167] https://github.com/STEllAR-GROUP/hpx/pull/3192
[1168] https://github.com/STEllAR-GROUP/hpx/pull/3159

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[1169] 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[1170] - shouldn't "-faligned-new" be a usage requirement?
- Issue #3841[1171] - Build error with msvc 19 caused by SFINAE and C++17
- Issue #3836[1172] - master branch does not build with idle rate counters enabled
- Issue #3819[1173] - Add debug suffix to modules built in debug mode
- Issue #3817[1174] - `HPX_INCLUDE_DIRS` contains non-existent directory
- Issue #3810[1175] - Source groups are not created for files in modules
- Issue #3805[1176] - HPX won't compile with `-DHPX_WITH_APEX=TRUE`
- Issue #3792[1177] - Barrier Hangs When Locality Zero not included
- Issue #3778[1178] - Replace `throw()` with `noexcept`
- Issue #3763[1179] - configurable sort limit per task
- Issue #3758[1180] - dataflow doesn't convert `future<future<T>>` to `future<T>`
- Issue #3757[1181] - When compiling undefined reference to `hpx::hpx_check_version_1_2` HPX V1.2.1, Ubuntu 18.04.01 Server Edition
- Issue #3753[1182] - `--hpx:list-counters=full` crashes

1169 https://www.cmake.org
1170 https://github.com/STEllAR-GROUP/hpx/issues/3863
1171 https://github.com/STEllAR-GROUP/hpx/issues/3841
1172 https://github.com/STEllAR-GROUP/hpx/issues/3836
1173 https://github.com/STEllAR-GROUP/hpx/issues/3819
1174 https://github.com/STEllAR-GROUP/hpx/issues/3817
1175 https://github.com/STEllAR-GROUP/hpx/issues/3810
1176 https://github.com/STEllAR-GROUP/hpx/issues/3805
1177 https://github.com/STEllAR-GROUP/hpx/issues/3792
1178 https://github.com/STEllAR-GROUP/hpx/issues/3778
1179 https://github.com/STEllAR-GROUP/hpx/issues/3763
1180 https://github.com/STEllAR-GROUP/hpx/issues/3758
1181 https://github.com/STEllAR-GROUP/hpx/issues/3757
1182 https://github.com/STEllAR-GROUP/hpx/issues/3753

- Issue #3746[1183] - Detection of MPI with pmix
- Issue #3744[1184] - Separate spinlock from same cacheline as internal data for all LCOs
- Issue #3743[1185] - hpxcxx's shebang doesn't specify the python version
- Issue #3738[1186] - Unable to debug parcelport on a single node
- Issue #3735[1187] - Latest master: Can't compile in MSVC
- Issue #3731[1188] - `util::bound` seems broken on Clang with older libstdc++
- Issue #3724[1189] - Allow to pre-set command line options through environment
- Issue #3723[1190] - examples/resource_partitioner build issue on master branch / ubuntu 18
- Issue #3721[1191] - faced a building error
- Issue #3720[1192] - Hello World example fails to link
- Issue #3719[1193] - pkg-config produces invalid output: `-l-pthread`
- Issue #3718[1194] - Please make the python executable configurable through cmake
- Issue #3717[1195] - interested to contribute to the organisation
- Issue #3699[1196] - Remove 'HPX runtime' executable
- Issue #3698[1197] - Ignore all locks while handling asserts
- Issue #3689[1198] - Incorrect and inconsistent website structure http://stellar.cct.lsu.edu/downloads/.
- Issue #3681[1199] - Broken links on http://stellar.cct.lsu.edu/2015/05/hpx-archives-now-on-gmane/
- Issue #3676[1200] - HPX master built from source, cmake fails to link main.cpp example in docs
- Issue #3673[1201] - HPX build fails with `std::atomic` missing error
- Issue #3670[1202] - Generate PDF again from documentation (with Sphinx)
- Issue #3643[1203] - Warnings when compiling HPX 1.2.1 with gcc 9
- Issue #3641[1204] - Trouble with using ranges-v3 and `hpx::parallel::reduce`
- Issue #3639[1205] - `util::unwrapping` does not work well with member functions

[1183] https://github.com/STEllAR-GROUP/hpx/issues/3746
[1184] https://github.com/STEllAR-GROUP/hpx/issues/3744
[1185] https://github.com/STEllAR-GROUP/hpx/issues/3743
[1186] https://github.com/STEllAR-GROUP/hpx/issues/3738
[1187] https://github.com/STEllAR-GROUP/hpx/issues/3735
[1188] https://github.com/STEllAR-GROUP/hpx/issues/3731
[1189] https://github.com/STEllAR-GROUP/hpx/issues/3724
[1190] https://github.com/STEllAR-GROUP/hpx/issues/3723
[1191] https://github.com/STEllAR-GROUP/hpx/issues/3721
[1192] https://github.com/STEllAR-GROUP/hpx/issues/3720
[1193] https://github.com/STEllAR-GROUP/hpx/issues/3719
[1194] https://github.com/STEllAR-GROUP/hpx/issues/3718
[1195] https://github.com/STEllAR-GROUP/hpx/issues/3717
[1196] https://github.com/STEllAR-GROUP/hpx/issues/3699
[1197] https://github.com/STEllAR-GROUP/hpx/issues/3698
[1198] https://github.com/STEllAR-GROUP/hpx/issues/3689
[1199] https://github.com/STEllAR-GROUP/hpx/issues/3681
[1200] https://github.com/STEllAR-GROUP/hpx/issues/3676
[1201] https://github.com/STEllAR-GROUP/hpx/issues/3673
[1202] https://github.com/STEllAR-GROUP/hpx/issues/3670
[1203] https://github.com/STEllAR-GROUP/hpx/issues/3643
[1204] https://github.com/STEllAR-GROUP/hpx/issues/3641
[1205] https://github.com/STEllAR-GROUP/hpx/issues/3639

- Issue #3634[1206] - The build fails if `shared_future<>::then` is called with a thread executor
- Issue #3622[1207] - VTune Amplifier 2019 not working with `use_itt_notify=1`
- Issue #3616[1208] - HPX Fails to Build with CUDA 10
- Issue #3612[1209] - False sharing of scheduling counters
- Issue #3609[1210] - executor_parameters timeout with gcc <= 7 and Debug mode
- Issue #3601[1211] - Misleading error message on power pc for rdtsc and rdtscp
- Issue #3598[1212] - Build of some examples fails when using Vc
- Issue #3594[1213] - Error: The number of OS threads requested (20) does not match the number of threads to bind (12): HPX(bad_parameter)
- Issue #3592[1214] - Undefined Reference Error
- Issue #3589[1215] - include could not find load file: HPX_Utils.cmake
- Issue #3587[1216] - HPX won't compile on POWER8 with Clang 7
- Issue #3583[1217] - Fedora and openSUSE instructions missing on "Distribution Packages" page
- Issue #3578[1218] - Build error when configuring with `HPX_HAVE_ALGORITHM_INPUT_ITERATOR_SUPPORT=ON`
- Issue #3575[1219] - Merge openSUSE reproducible patch
- Issue #3570[1220] - Update HPX to work with the latest VC version
- Issue #3567[1221] - Build succeed and make failed for `hpx:cout`
- Issue #3565[1222] - Polymorphic simple component destructor not getting called
- Issue #3559[1223] - 1.2.0 is missing from download page
- Issue #3554[1224] - Clang 6.0 warning of hiding overloaded virtual function
- Issue #3510[1225] - Build on ppc64 fails
- Issue #3482[1226] - Improve error message when `HPX_WITH_MAX_CPU_COUNT` is too low for given system
- Issue #3453[1227] - Two HPX applications can't run at the same time.
- Issue #3452[1228] - Scaling issue on the change to 2 NUMA domains

[1206] https://github.com/STEllAR-GROUP/hpx/issues/3634
[1207] https://github.com/STEllAR-GROUP/hpx/issues/3622
[1208] https://github.com/STEllAR-GROUP/hpx/issues/3616
[1209] https://github.com/STEllAR-GROUP/hpx/issues/3612
[1210] https://github.com/STEllAR-GROUP/hpx/issues/3609
[1211] https://github.com/STEllAR-GROUP/hpx/issues/3601
[1212] https://github.com/STEllAR-GROUP/hpx/issues/3598
[1213] https://github.com/STEllAR-GROUP/hpx/issues/3594
[1214] https://github.com/STEllAR-GROUP/hpx/issues/3592
[1215] https://github.com/STEllAR-GROUP/hpx/issues/3589
[1216] https://github.com/STEllAR-GROUP/hpx/issues/3587
[1217] https://github.com/STEllAR-GROUP/hpx/issues/3583
[1218] https://github.com/STEllAR-GROUP/hpx/issues/3578
[1219] https://github.com/STEllAR-GROUP/hpx/issues/3575
[1220] https://github.com/STEllAR-GROUP/hpx/issues/3570
[1221] https://github.com/STEllAR-GROUP/hpx/issues/3567
[1222] https://github.com/STEllAR-GROUP/hpx/issues/3565
[1223] https://github.com/STEllAR-GROUP/hpx/issues/3559
[1224] https://github.com/STEllAR-GROUP/hpx/issues/3554
[1225] https://github.com/STEllAR-GROUP/hpx/issues/3510
[1226] https://github.com/STEllAR-GROUP/hpx/issues/3482
[1227] https://github.com/STEllAR-GROUP/hpx/issues/3453
[1228] https://github.com/STEllAR-GROUP/hpx/issues/3452

- Issue #3442[1229] - HPX set_difference, set_intersection failure cases
- Issue #3437[1230] - Ensure parent_task pointer when child task is created and child/parent are on same locality
- Issue #3255[1231] - Suspension with lock for `--hpx:list-component-types`
- Issue #3034[1232] - Use C++17 structured bindings for serialization
- Issue #2999[1233] - Change thread scheduling use of `size_t` for thread indexing

Closed pull requests

- PR #3865[1234] - adds hpx_target_compile_option_if_available
- PR #3864[1235] - Helper functions that are useful in numa binding and testing of allocator
- PR #3862[1236] - Temporary fix to local_dataflow_boost_small_vector test
- PR #3860[1237] - Add cache line padding to intermediate results in for loop reduction
- PR #3859[1238] - Remove HPX_TLL_PUBLIC and HPX_TLL_PRIVATE from CMake files
- PR #3858[1239] - Add compile flags and definitions to modules
- PR #3851[1240] - update hpxmp release tag to v0.2.0
- PR #3849[1241] - Correct BOOST_ROOT variable name in quick start guide
- PR #3847[1242] - Fix attach_debugger configuration option
- PR #3846[1243] - Add tests for libs header tests
- PR #3844[1244] - Fixing source_groups in preprocessor module to properly handle compatibility headers
- PR #3843[1245] - This fixes the launch_process/launched_process pair of tests
- PR #3842[1246] - Fix macro call with ITTNOTIFY enabled
- PR #3840[1247] - Fixing SLURM environment parsing
- PR #3837[1248] - Fixing misplaced #endif
- PR #3835[1249] - make all latch members protected for consistency
- PR #3834[1250] - Disable transpose_block_numa example on CircleCI

1229 https://github.com/STEllAR-GROUP/hpx/issues/3442
1230 https://github.com/STEllAR-GROUP/hpx/issues/3437
1231 https://github.com/STEllAR-GROUP/hpx/issues/3255
1232 https://github.com/STEllAR-GROUP/hpx/issues/3034
1233 https://github.com/STEllAR-GROUP/hpx/issues/2999
1234 https://github.com/STEllAR-GROUP/hpx/pull/3865
1235 https://github.com/STEllAR-GROUP/hpx/pull/3864
1236 https://github.com/STEllAR-GROUP/hpx/pull/3862
1237 https://github.com/STEllAR-GROUP/hpx/pull/3860
1238 https://github.com/STEllAR-GROUP/hpx/pull/3859
1239 https://github.com/STEllAR-GROUP/hpx/pull/3858
1240 https://github.com/STEllAR-GROUP/hpx/pull/3851
1241 https://github.com/STEllAR-GROUP/hpx/pull/3849
1242 https://github.com/STEllAR-GROUP/hpx/pull/3847
1243 https://github.com/STEllAR-GROUP/hpx/pull/3846
1244 https://github.com/STEllAR-GROUP/hpx/pull/3844
1245 https://github.com/STEllAR-GROUP/hpx/pull/3843
1246 https://github.com/STEllAR-GROUP/hpx/pull/3842
1247 https://github.com/STEllAR-GROUP/hpx/pull/3840
1248 https://github.com/STEllAR-GROUP/hpx/pull/3837
1249 https://github.com/STEllAR-GROUP/hpx/pull/3835
1250 https://github.com/STEllAR-GROUP/hpx/pull/3834

- PR #3833[1251] - make latch **counter_** protected for deriving latch in hpxmp
- PR #3831[1252] - Fix CircleCI config for modules
- PR #3830[1253] - minor fix: option HPX_WITH_TEST was not working correctly
- PR #3828[1254] - Avoid for binaries that depend on HPX to directly link against internal modules
- PR #3827[1255] - Adding shortcut for `hpx::get_ptr<>(sync, id)` for a local, non-migratable objects
- PR #3826[1256] - Fix and update modules documentation
- PR #3825[1257] - Updating default APEX version to 2.1.3 with HPX
- PR #3823[1258] - Fix pkgconfig libs handling
- PR #3822[1259] - Change includes in hpx_wrap.cpp to more specific includes
- PR #3821[1260] - Disable barrier_3792 test when networking is disabled
- PR #3820[1261] - Assorted CMake fixes
- PR #3815[1262] - Removing left-over debug output
- PR #3814[1263] - Allow setting default scheduler mode via the configuration database
- PR #3813[1264] - Make the deprecation warnings issued by the old pp headers optional
- PR #3812[1265] - Windows requires to handle symlinks to directories differently from those linking files
- PR #3811[1266] - Clean up PP module and library skeleton
- PR #3806[1267] - Moving include path configuration to before APEX
- PR #3804[1268] - Fix latch
- PR #3803[1269] - Update hpxcxx to look at lib64 and use python3
- PR #3802[1270] - Numa binding allocator
- PR #3801[1271] - Remove duplicated includes
- PR #3800[1272] - Attempt to fix Posix context switching after lazy init changes
- PR #3798[1273] - count and count_if accepts different iterator types

[1251] https://github.com/STEllAR-GROUP/hpx/pull/3833
[1252] https://github.com/STEllAR-GROUP/hpx/pull/3831
[1253] https://github.com/STEllAR-GROUP/hpx/pull/3830
[1254] https://github.com/STEllAR-GROUP/hpx/pull/3828
[1255] https://github.com/STEllAR-GROUP/hpx/pull/3827
[1256] https://github.com/STEllAR-GROUP/hpx/pull/3826
[1257] https://github.com/STEllAR-GROUP/hpx/pull/3825
[1258] https://github.com/STEllAR-GROUP/hpx/pull/3823
[1259] https://github.com/STEllAR-GROUP/hpx/pull/3822
[1260] https://github.com/STEllAR-GROUP/hpx/pull/3821
[1261] https://github.com/STEllAR-GROUP/hpx/pull/3820
[1262] https://github.com/STEllAR-GROUP/hpx/pull/3815
[1263] https://github.com/STEllAR-GROUP/hpx/pull/3814
[1264] https://github.com/STEllAR-GROUP/hpx/pull/3813
[1265] https://github.com/STEllAR-GROUP/hpx/pull/3812
[1266] https://github.com/STEllAR-GROUP/hpx/pull/3811
[1267] https://github.com/STEllAR-GROUP/hpx/pull/3806
[1268] https://github.com/STEllAR-GROUP/hpx/pull/3804
[1269] https://github.com/STEllAR-GROUP/hpx/pull/3803
[1270] https://github.com/STEllAR-GROUP/hpx/pull/3802
[1271] https://github.com/STEllAR-GROUP/hpx/pull/3801
[1272] https://github.com/STEllAR-GROUP/hpx/pull/3800
[1273] https://github.com/STEllAR-GROUP/hpx/pull/3798

- PR #3797[1274] - Adding a couple of `override` keywords to overloaded virtual functions
- PR #3796[1275] - Re-enable testing all schedulers in shutdown_suspended_test
- PR #3795[1276] - Change `std::terminate` to std::abort in `SIGSEGV` handler
- PR #3794[1277] - Fixing #3792
- PR #3793[1278] - Extending migrate_polymorphic_component unit test
- PR #3791[1279] - Change `throw()` to `noexcept`
- PR #3790[1280] - Remove deprecated options for 1.3.0 release
- PR #3789[1281] - Remove Boost filesystem compatibility header
- PR #3788[1282] - Disabled even more spots that should not execute if networking is disabled
- PR #3787[1283] - Bump minimal boost supported version to 1.61.0
- PR #3786[1284] - Bump minimum required versions for 1.3.0 release
- PR #3785[1285] - Explicitly set number of jobs for all ninja invocations on CircleCI
- PR #3784[1286] - Fix leak and address sanitizer problems
- PR #3783[1287] - Disabled even more spots that should not execute is networking is disabled
- PR #3782[1288] - Cherry-picked tuple and thread_init_data fixes from #3701
- PR #3781[1289] - Fix generic context coroutines after lazy stack allocation changes
- PR #3780[1290] - Rename hello world examples
- PR #3776[1291] - Sort algorithms now use the supplied chunker to determine the required minimal chunk size
- PR #3775[1292] - Disable Boost auto-linking
- PR #3774[1293] - Tag and push stable builds
- PR #3773[1294] - Enable migration of polymorphic components
- PR #3771[1295] - Fix link to stackoverflow in documentation
- PR #3770[1296] - Replacing constexpr if in brace-serialization code

1274 https://github.com/STEllAR-GROUP/hpx/pull/3797
1275 https://github.com/STEllAR-GROUP/hpx/pull/3796
1276 https://github.com/STEllAR-GROUP/hpx/pull/3795
1277 https://github.com/STEllAR-GROUP/hpx/pull/3794
1278 https://github.com/STEllAR-GROUP/hpx/pull/3793
1279 https://github.com/STEllAR-GROUP/hpx/pull/3791
1280 https://github.com/STEllAR-GROUP/hpx/pull/3790
1281 https://github.com/STEllAR-GROUP/hpx/pull/3789
1282 https://github.com/STEllAR-GROUP/hpx/pull/3788
1283 https://github.com/STEllAR-GROUP/hpx/pull/3787
1284 https://github.com/STEllAR-GROUP/hpx/pull/3786
1285 https://github.com/STEllAR-GROUP/hpx/pull/3785
1286 https://github.com/STEllAR-GROUP/hpx/pull/3784
1287 https://github.com/STEllAR-GROUP/hpx/pull/3783
1288 https://github.com/STEllAR-GROUP/hpx/pull/3782
1289 https://github.com/STEllAR-GROUP/hpx/pull/3781
1290 https://github.com/STEllAR-GROUP/hpx/pull/3780
1291 https://github.com/STEllAR-GROUP/hpx/pull/3776
1292 https://github.com/STEllAR-GROUP/hpx/pull/3775
1293 https://github.com/STEllAR-GROUP/hpx/pull/3774
1294 https://github.com/STEllAR-GROUP/hpx/pull/3773
1295 https://github.com/STEllAR-GROUP/hpx/pull/3771
1296 https://github.com/STEllAR-GROUP/hpx/pull/3770

- PR #3769[1297] - Fix SIGSEGV handler
- PR #3768[1298] - Adding flags to scheduler allowing to control thread stealing and idle back-off
- PR #3767[1299] - Fix help formatting in hpxrun.py
- PR #3765[1300] - Fix a couple of bugs in the thread test
- PR #3764[1301] - Workaround for SFINAE regression in msvc14.2
- PR #3762[1302] - Prevent MSVC from prematurely instantiating things
- PR #3761[1303] - Update python scripts to work with python 3
- PR #3760[1304] - Fix callable vtable for GCC4.9
- PR #3759[1305] - Rename `PAGE_SIZE` to `PAGE_SIZE_` because AppleClang
- PR #3755[1306] - Making sure locks are not held during suspension
- PR #3754[1307] - Disable more code if networking is not available/not enabled
- PR #3752[1308] - Move `util::format` implementation to source file
- PR #3751[1309] - Fixing problems with `lcos::barrier` and iostreams
- PR #3750[1310] - Change error message to take into account `use_guard_page` setting
- PR #3749[1311] - Fix lifetime problem in `run_as_hpx_thread`
- PR #3748[1312] - Fixed unusable behavior of the clang code analyzer.
- PR #3747[1313] - Added `PMIX_RANK` to the defaults of `HPX_WITH_PARCELPORT_MPI_ENV`.
- PR #3745[1314] - Introduced `cache_aligned_data` and `cache_line_data` helper structure
- PR #3742[1315] - Remove more unused functionality from util/logging
- PR #3740[1316] - Fix includes in partitioned vector tests
- PR #3739[1317] - More fixes to make sure that `std::flush` really flushes all output
- PR #3737[1318] - Fix potential shutdown problems
- PR #3736[1319] - Fix `guided_pool_executor` after dataflow changes caused compilation fail

[1297] https://github.com/STEllAR-GROUP/hpx/pull/3769
[1298] https://github.com/STEllAR-GROUP/hpx/pull/3768
[1299] https://github.com/STEllAR-GROUP/hpx/pull/3767
[1300] https://github.com/STEllAR-GROUP/hpx/pull/3765
[1301] https://github.com/STEllAR-GROUP/hpx/pull/3764
[1302] https://github.com/STEllAR-GROUP/hpx/pull/3762
[1303] https://github.com/STEllAR-GROUP/hpx/pull/3761
[1304] https://github.com/STEllAR-GROUP/hpx/pull/3760
[1305] https://github.com/STEllAR-GROUP/hpx/pull/3759
[1306] https://github.com/STEllAR-GROUP/hpx/pull/3755
[1307] https://github.com/STEllAR-GROUP/hpx/pull/3754
[1308] https://github.com/STEllAR-GROUP/hpx/pull/3752
[1309] https://github.com/STEllAR-GROUP/hpx/pull/3751
[1310] https://github.com/STEllAR-GROUP/hpx/pull/3750
[1311] https://github.com/STEllAR-GROUP/hpx/pull/3749
[1312] https://github.com/STEllAR-GROUP/hpx/pull/3748
[1313] https://github.com/STEllAR-GROUP/hpx/pull/3747
[1314] https://github.com/STEllAR-GROUP/hpx/pull/3745
[1315] https://github.com/STEllAR-GROUP/hpx/pull/3742
[1316] https://github.com/STEllAR-GROUP/hpx/pull/3740
[1317] https://github.com/STEllAR-GROUP/hpx/pull/3739
[1318] https://github.com/STEllAR-GROUP/hpx/pull/3737
[1319] https://github.com/STEllAR-GROUP/hpx/pull/3736

- PR #3734[1320] - Limiting executor
- PR #3732[1321] - More constrained bound constructors
- PR #3730[1322] - Attempt to fix deadlocks during component loading
- PR #3729[1323] - Add latch member function `count_up` and reset, requested by hpxMP
- PR #3728[1324] - Send even empty buffers on `hpx::endl` and `hpx::flush`
- PR #3727[1325] - Adding example demonstrating how to customize the memory management for a component
- PR #3726[1326] - Adding support for passing command line options through the `HPX_COMMANDLINE_OPTIONS` environment variable
- PR #3722[1327] - Document known broken OpenMPI builds
- PR #3716[1328] - Add barrier reset function, requested by hpxMP for reusing barrier
- PR #3715[1329] - More work on functions and vtables
- PR #3714[1330] - Generate single-page HTML, PDF, manpage from documentation
- PR #3713[1331] - Updating default APEX version to 2.1.2
- PR #3712[1332] - Update release procedure
- PR #3710[1333] - Fix the C++11 build, after #3704
- PR #3709[1334] - Move some component_registry functionality to source file
- PR #3708[1335] - Ignore all locks while handling assertions
- PR #3707[1336] - Remove obsolete hpx runtime executable
- PR #3705[1337] - Fix and simplify `make_ready_future` overload sets
- PR #3704[1338] - Reduce use of binders
- PR #3703[1339] - Ini
- PR #3702[1340] - Fixing CUDA compiler errors
- PR #3700[1341] - Added `barrier::increment` function to increase total number of thread
- PR #3697[1342] - One more attempt to fix migration…

1320 https://github.com/STEllAR-GROUP/hpx/pull/3734
1321 https://github.com/STEllAR-GROUP/hpx/pull/3732
1322 https://github.com/STEllAR-GROUP/hpx/pull/3730
1323 https://github.com/STEllAR-GROUP/hpx/pull/3729
1324 https://github.com/STEllAR-GROUP/hpx/pull/3728
1325 https://github.com/STEllAR-GROUP/hpx/pull/3727
1326 https://github.com/STEllAR-GROUP/hpx/pull/3726
1327 https://github.com/STEllAR-GROUP/hpx/pull/3722
1328 https://github.com/STEllAR-GROUP/hpx/pull/3716
1329 https://github.com/STEllAR-GROUP/hpx/pull/3715
1330 https://github.com/STEllAR-GROUP/hpx/pull/3714
1331 https://github.com/STEllAR-GROUP/hpx/pull/3713
1332 https://github.com/STEllAR-GROUP/hpx/pull/3712
1333 https://github.com/STEllAR-GROUP/hpx/pull/3710
1334 https://github.com/STEllAR-GROUP/hpx/pull/3709
1335 https://github.com/STEllAR-GROUP/hpx/pull/3708
1336 https://github.com/STEllAR-GROUP/hpx/pull/3707
1337 https://github.com/STEllAR-GROUP/hpx/pull/3705
1338 https://github.com/STEllAR-GROUP/hpx/pull/3704
1339 https://github.com/STEllAR-GROUP/hpx/pull/3703
1340 https://github.com/STEllAR-GROUP/hpx/pull/3702
1341 https://github.com/STEllAR-GROUP/hpx/pull/3700
1342 https://github.com/STEllAR-GROUP/hpx/pull/3697

- PR #3694[1343] - Fixing component migration
- PR #3693[1344] - Print thread state when getting disallowed value in set_thread_state
- PR #3692[1345] - Only disable `constexpr` with clang-cuda, not nvcc+gcc
- PR #3691[1346] - Link with libsupc++ if needed for thread_local
- PR #3690[1347] - Remove thousands separators in set_operations_3442 to comply with C++11
- PR #3688[1348] - Decouple serialization from function vtables
- PR #3687[1349] - Fix a couple of test failures
- PR #3686[1350] - Make sure tests.unit.build are run after install on CircleCI
- PR #3685[1351] - Revise quickstart CMakeLists.txt explanation
- PR #3684[1352] - Provide concept emulation for Ranges-TS concepts
- PR #3683[1353] - Ignore uninitialized chunks
- PR #3682[1354] - Ignore uninitialized chunks. Check proper indices.
- PR #3680[1355] - Ignore uninitialized chunks. Check proper range indices
- PR #3679[1356] - Simplify basic action implementations
- PR #3678[1357] - Making sure `HPX_HAVE_LIBATOMIC` is unset before checking
- PR #3677[1358] - Fix generated full version number to be usable in expressions
- PR #3674[1359] - Reduce functional utilities call depth
- PR #3672[1360] - Change new build system to use existing macros related to pseudo dependencies
- PR #3669[1361] - Remove indirection in `function_ref` when thread description is disabled
- PR #3668[1362] - Unbreaking `async_*cb*` tests
- PR #3667[1363] - Generate version.hpp
- PR #3665[1364] - Enabling MPI parcelport for gitlab runners
- PR #3664[1365] - making clang-tidy work properly again

1343 https://github.com/STEllAR-GROUP/hpx/pull/3694
1344 https://github.com/STEllAR-GROUP/hpx/pull/3693
1345 https://github.com/STEllAR-GROUP/hpx/pull/3692
1346 https://github.com/STEllAR-GROUP/hpx/pull/3691
1347 https://github.com/STEllAR-GROUP/hpx/pull/3690
1348 https://github.com/STEllAR-GROUP/hpx/pull/3688
1349 https://github.com/STEllAR-GROUP/hpx/pull/3687
1350 https://github.com/STEllAR-GROUP/hpx/pull/3686
1351 https://github.com/STEllAR-GROUP/hpx/pull/3685
1352 https://github.com/STEllAR-GROUP/hpx/pull/3684
1353 https://github.com/STEllAR-GROUP/hpx/pull/3683
1354 https://github.com/STEllAR-GROUP/hpx/pull/3682
1355 https://github.com/STEllAR-GROUP/hpx/pull/3680
1356 https://github.com/STEllAR-GROUP/hpx/pull/3679
1357 https://github.com/STEllAR-GROUP/hpx/pull/3678
1358 https://github.com/STEllAR-GROUP/hpx/pull/3677
1359 https://github.com/STEllAR-GROUP/hpx/pull/3674
1360 https://github.com/STEllAR-GROUP/hpx/pull/3672
1361 https://github.com/STEllAR-GROUP/hpx/pull/3669
1362 https://github.com/STEllAR-GROUP/hpx/pull/3668
1363 https://github.com/STEllAR-GROUP/hpx/pull/3667
1364 https://github.com/STEllAR-GROUP/hpx/pull/3665
1365 https://github.com/STEllAR-GROUP/hpx/pull/3664

- PR #3662[1366] - Attempt to fix exception handling
- PR #3661[1367] - Move `lcos::latch` to source file
- PR #3660[1368] - Fix accidentally explicit gid_type default constructor
- PR #3659[1369] - Parallel executor latch
- PR #3658[1370] - Fixing execution_parameters
- PR #3657[1371] - Avoid dangling references in wait_all
- PR #3656[1372] - Avoiding lifetime problems with sync_put_parcel
- PR #3655[1373] - Fixing nullptr dereference inside of function
- PR #3652[1374] - Attempt to fix `thread_map_type` definition with C++11
- PR #3650[1375] - Allowing for end iterator being different from begin iterator
- PR #3649[1376] - Added architecture identification to cmake to be able to detect timestamp support
- PR #3645[1377] - Enabling sanitizers on gitlab runner
- PR #3644[1378] - Attempt to tackle timeouts during startup
- PR #3642[1379] - Cleanup parallel partitioners
- PR #3640[1380] - Dataflow now works with functions that return a reference
- PR #3637[1381] - Merging the executor-enabled overloads of `shared_future<>::then`
- PR #3633[1382] - Replace deprecated boost endian macros
- PR #3632[1383] - Add instructions on getting HPX to documentation
- PR #3631[1384] - Simplify parcel creation
- PR #3630[1385] - Small additions and fixes to release procedure
- PR #3629[1386] - Modular pp
- PR #3627[1387] - Implement `util::function_ref`
- PR #3626[1388] - Fix cancelable_action_client example

1366 https://github.com/STEllAR-GROUP/hpx/pull/3662
1367 https://github.com/STEllAR-GROUP/hpx/pull/3661
1368 https://github.com/STEllAR-GROUP/hpx/pull/3660
1369 https://github.com/STEllAR-GROUP/hpx/pull/3659
1370 https://github.com/STEllAR-GROUP/hpx/pull/3658
1371 https://github.com/STEllAR-GROUP/hpx/pull/3657
1372 https://github.com/STEllAR-GROUP/hpx/pull/3656
1373 https://github.com/STEllAR-GROUP/hpx/pull/3655
1374 https://github.com/STEllAR-GROUP/hpx/pull/3652
1375 https://github.com/STEllAR-GROUP/hpx/pull/3650
1376 https://github.com/STEllAR-GROUP/hpx/pull/3649
1377 https://github.com/STEllAR-GROUP/hpx/pull/3645
1378 https://github.com/STEllAR-GROUP/hpx/pull/3644
1379 https://github.com/STEllAR-GROUP/hpx/pull/3642
1380 https://github.com/STEllAR-GROUP/hpx/pull/3640
1381 https://github.com/STEllAR-GROUP/hpx/pull/3637
1382 https://github.com/STEllAR-GROUP/hpx/pull/3633
1383 https://github.com/STEllAR-GROUP/hpx/pull/3632
1384 https://github.com/STEllAR-GROUP/hpx/pull/3631
1385 https://github.com/STEllAR-GROUP/hpx/pull/3630
1386 https://github.com/STEllAR-GROUP/hpx/pull/3629
1387 https://github.com/STEllAR-GROUP/hpx/pull/3627
1388 https://github.com/STEllAR-GROUP/hpx/pull/3626

- PR #3625[1389] - Added automatic serialization for simple structs (see #3034)
- PR #3624[1390] - Updating the default order of priority for `thread_description`
- PR #3621[1391] - Update copyright year and other small formatting fixes
- PR #3620[1392] - Adding support for gitlab runner
- PR #3619[1393] - Store debug logs and core dumps on CircleCI
- PR #3618[1394] - Various optimizations
- PR #3617[1395] - Fix link to the gpg key (#2)
- PR #3615[1396] - Fix unused variable warnings with networking off
- PR #3614[1397] - Restructuring counter data in scheduler to reduce false sharing
- PR #3613[1398] - Adding support for gitlab runners
- PR #3610[1399] - Don't wait for `stop_condition` in main thread
- PR #3608[1400] - Add inline keyword to `invalid_thread_id` definition for nvcc
- PR #3607[1401] - Adding configuration key that allows one to explicitly add a directory to the component search path
- PR #3606[1402] - Add nvcc to exclude constexpress since is it not supported by nvcc
- PR #3605[1403] - Add `inline` to definition of checkpoint stream operators to fix link error
- PR #3604[1404] - Use format for string formatting
- PR #3603[1405] - Improve the error message for using to less `MAX_CPU_COUNT`
- PR #3602[1406] - Improve the error message for to small values of `MAX_CPU_COUNT`
- PR #3600[1407] - Parallel executor aggregated
- PR #3599[1408] - Making sure networking is disabled for default one-locality-runs
- PR #3596[1409] - Store thread exit functions in `forward_list` instead of `deque` to avoid allocations
- PR #3590[1410] - Fix typo/mistake in thread queue `cleanup_terminated`
- PR #3588[1411] - Fix formatting errors in launching_and_configuring_hpx_applications.rst

[1389] https://github.com/STEllAR-GROUP/hpx/pull/3625
[1390] https://github.com/STEllAR-GROUP/hpx/pull/3624
[1391] https://github.com/STEllAR-GROUP/hpx/pull/3621
[1392] https://github.com/STEllAR-GROUP/hpx/pull/3620
[1393] https://github.com/STEllAR-GROUP/hpx/pull/3619
[1394] https://github.com/STEllAR-GROUP/hpx/pull/3618
[1395] https://github.com/STEllAR-GROUP/hpx/pull/3617
[1396] https://github.com/STEllAR-GROUP/hpx/pull/3615
[1397] https://github.com/STEllAR-GROUP/hpx/pull/3614
[1398] https://github.com/STEllAR-GROUP/hpx/pull/3613
[1399] https://github.com/STEllAR-GROUP/hpx/pull/3610
[1400] https://github.com/STEllAR-GROUP/hpx/pull/3608
[1401] https://github.com/STEllAR-GROUP/hpx/pull/3607
[1402] https://github.com/STEllAR-GROUP/hpx/pull/3606
[1403] https://github.com/STEllAR-GROUP/hpx/pull/3605
[1404] https://github.com/STEllAR-GROUP/hpx/pull/3604
[1405] https://github.com/STEllAR-GROUP/hpx/pull/3603
[1406] https://github.com/STEllAR-GROUP/hpx/pull/3602
[1407] https://github.com/STEllAR-GROUP/hpx/pull/3600
[1408] https://github.com/STEllAR-GROUP/hpx/pull/3599
[1409] https://github.com/STEllAR-GROUP/hpx/pull/3596
[1410] https://github.com/STEllAR-GROUP/hpx/pull/3590
[1411] https://github.com/STEllAR-GROUP/hpx/pull/3588

- PR #3586[1412] - Make bind propagate value category
- PR #3585[1413] - Extend Cmake for building hpx as distribution packages (refs #3575)
- PR #3584[1414] - Untangle function storage from object pointer
- PR #3582[1415] - Towards Modularized HPX
- PR #3580[1416] - Remove extra `||` in merge.hpp
- PR #3577[1417] - Partially revert "Remove vtable empty flag"
- PR #3576[1418] - Make sure empty startup/shutdown functions are not being used
- PR #3574[1419] - Make sure `DATAPAR` settings are conveyed to depending projects
- PR #3573[1420] - Make sure HPX is usable with latest released version of Vc (V1.4.1)
- PR #3572[1421] - Adding test ensuring ticket 3565 is fixed
- PR #3571[1422] - Make empty `[unique_]function` vtable non-dependent
- PR #3566[1423] - Fix compilation with dynamic bitset for CPU masks
- PR #3563[1424] - Drop `util::[unique_]function` target_type
- PR #3562[1425] - Removing the target suffixes
- PR #3561[1426] - Replace executor traits return type deduction (keep non-SFINAE)
- PR #3557[1427] - Replace the last usages of boost::atomic
- PR #3556[1428] - Replace `boost::scoped_array` with `std::unique_ptr`
- PR #3552[1429] - (Re)move APEX readme
- PR #3548[1430] - Replace `boost::scoped_ptr` with `std::unique_ptr`
- PR #3547[1431] - Remove last use of Boost.Signals2
- PR #3544[1432] - Post 1.2.0 version bumps
- PR #3543[1433] - added Ubuntu dependency list to readme
- PR #3531[1434] - Warnings, warnings...

1412 https://github.com/STEllAR-GROUP/hpx/pull/3586
1413 https://github.com/STEllAR-GROUP/hpx/pull/3585
1414 https://github.com/STEllAR-GROUP/hpx/pull/3584
1415 https://github.com/STEllAR-GROUP/hpx/pull/3582
1416 https://github.com/STEllAR-GROUP/hpx/pull/3580
1417 https://github.com/STEllAR-GROUP/hpx/pull/3577
1418 https://github.com/STEllAR-GROUP/hpx/pull/3576
1419 https://github.com/STEllAR-GROUP/hpx/pull/3574
1420 https://github.com/STEllAR-GROUP/hpx/pull/3573
1421 https://github.com/STEllAR-GROUP/hpx/pull/3572
1422 https://github.com/STEllAR-GROUP/hpx/pull/3571
1423 https://github.com/STEllAR-GROUP/hpx/pull/3566
1424 https://github.com/STEllAR-GROUP/hpx/pull/3563
1425 https://github.com/STEllAR-GROUP/hpx/pull/3562
1426 https://github.com/STEllAR-GROUP/hpx/pull/3561
1427 https://github.com/STEllAR-GROUP/hpx/pull/3557
1428 https://github.com/STEllAR-GROUP/hpx/pull/3556
1429 https://github.com/STEllAR-GROUP/hpx/pull/3552
1430 https://github.com/STEllAR-GROUP/hpx/pull/3548
1431 https://github.com/STEllAR-GROUP/hpx/pull/3547
1432 https://github.com/STEllAR-GROUP/hpx/pull/3544
1433 https://github.com/STEllAR-GROUP/hpx/pull/3543
1434 https://github.com/STEllAR-GROUP/hpx/pull/3531

- PR #3527[1435] - Add CircleCI filter for building all tags
- PR #3525[1436] - Segmented algorithms
- PR #3517[1437] - Replace `boost::regex` with C++11 `<regex>`
- PR #3514[1438] - Cleaning up the build system
- PR #3505[1439] - Fixing type attribute warning for `transfer_action`
- PR #3504[1440] - Add support for rpm packaging
- PR #3499[1441] - Improving spinlock pools
- PR #3498[1442] - Remove thread specific ptr
- PR #3486[1443] - Fix comparison for expect_connecting_localities config entry
- PR #3469[1444] - Enable (existing) code for extracting stack pointer on Power platform

2.11.6 *HPX* V1.2.1 (Feb 19, 2019)

General changes

This is a bugfix release. It contains the following changes:

- Fix compilation on ARM, s390x and 32-bit architectures.
- Fix a critical bug in the `future` implementation.
- Fix several problems in the CMake configuration which affects external projects.
- Add support for Boost 1.69.0.

Closed issues

- Issue #3638[1445] - Build HPX 1.2 with boost 1.69
- Issue #3635[1446] - Non-deterministic crashing on Stampede2
- Issue #3550[1447] - 1>e:000workhpxsrcthrow_exception.cpp(54): error C2440: '<function-style-cast>': cannot convert from 'boost::system::error_code' to 'hpx::exception'
- Issue #3549[1448] - HPX 1.2.0 does not build on i686, but release candidate did
- Issue #3511[1449] - Build on s390x fails
- Issue #3509[1450] - Build on armv7l fails

[1435] https://github.com/STEllAR-GROUP/hpx/pull/3527
[1436] https://github.com/STEllAR-GROUP/hpx/pull/3525
[1437] https://github.com/STEllAR-GROUP/hpx/pull/3517
[1438] https://github.com/STEllAR-GROUP/hpx/pull/3514
[1439] https://github.com/STEllAR-GROUP/hpx/pull/3505
[1440] https://github.com/STEllAR-GROUP/hpx/pull/3504
[1441] https://github.com/STEllAR-GROUP/hpx/pull/3499
[1442] https://github.com/STEllAR-GROUP/hpx/pull/3498
[1443] https://github.com/STEllAR-GROUP/hpx/pull/3486
[1444] https://github.com/STEllAR-GROUP/hpx/pull/3469
[1445] https://github.com/STEllAR-GROUP/hpx/issues/3638
[1446] https://github.com/STEllAR-GROUP/hpx/issues/3635
[1447] https://github.com/STEllAR-GROUP/hpx/issues/3550
[1448] https://github.com/STEllAR-GROUP/hpx/issues/3549
[1449] https://github.com/STEllAR-GROUP/hpx/issues/3511
[1450] https://github.com/STEllAR-GROUP/hpx/issues/3509

Closed pull requests

- PR #3695[1451] - Don't install CMake templates and packaging files
- PR #3666[1452] - Fixing yet another race in future_data
- PR #3663[1453] - Fixing race between setting and getting the value inside future_data
- PR #3648[1454] - Adding timestamp option for S390x platform
- PR #3647[1455] - Blind attempt to fix warnings issued by gcc V9
- PR #3611[1456] - Include GNUInstallDirs earlier to have it available for subdirectories
- PR #3595[1457] - Use GNUInstallDirs lib path in pkgconfig config file
- PR #3593[1458] - Add include(GNUInstallDirs) to HPXMacros.cmake
- PR #3591[1459] - Fix compilation error on arm7 architecture. Compiles and runs on Fedora 29 on Pi 3.
- PR #3558[1460] - Adding constructor *exception(boost::system::error_code const&)*
- PR #3555[1461] - cmake: make install locations configurable
- PR #3551[1462] - Fix uint64_t causing compilation fail on i686

2.11.7 *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[1463] or pkg-config. The old behaviour can be restored by setting `HPX_WITH_DYNAMIC_HPX_MAIN=OFF` during CMake[1464] 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.

[1451] https://github.com/STEllAR-GROUP/hpx/pull/3695
[1452] https://github.com/STEllAR-GROUP/hpx/pull/3666
[1453] https://github.com/STEllAR-GROUP/hpx/pull/3663
[1454] https://github.com/STEllAR-GROUP/hpx/pull/3648
[1455] https://github.com/STEllAR-GROUP/hpx/pull/3647
[1456] https://github.com/STEllAR-GROUP/hpx/pull/3611
[1457] https://github.com/STEllAR-GROUP/hpx/pull/3595
[1458] https://github.com/STEllAR-GROUP/hpx/pull/3593
[1459] https://github.com/STEllAR-GROUP/hpx/pull/3591
[1460] https://github.com/STEllAR-GROUP/hpx/pull/3558
[1461] https://github.com/STEllAR-GROUP/hpx/pull/3555
[1462] https://github.com/STEllAR-GROUP/hpx/pull/3551
[1463] https://www.cmake.org
[1464] https://www.cmake.org

- 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[1465] 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[1466] 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[1467] configuration.
- It is now possible to register callback functions for *HPX* worker thread events. Callbacks can be registered for starting and stopping worker threads, and for when errors occur.

Breaking changes

- The implementation of `hpx_main.hpp` has changed. If you are using custom Makefiles you will need to make changes. Please see the documentation on *using Makefiles* for more details.
- The default value of `--hpx:threads` has changed from `all` to `cores`. The new option `cores` only starts one worker thread per core.
- We have dropped support for Boost 1.56 and 1.57. The minimal version of Boost we now test is 1.58.
- Our `boost::format`-based formatting implementation has been revised and replaced with a custom implementation. This changes the formatting syntax and requires changes if you are relying on `hpx::util::format` or `hpx::util::format_to`. The pull request for this change contains more information: PR #3266[1468].
- 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`.

[1465] https://www.cmake.org
[1466] https://www.cmake.org
[1467] https://www.cmake.org
[1468] https://github.com/STEllAR-GROUP/hpx/pull/3266

Closed issues

- Issue #3538[1469] - numa handling incorrect for hwloc 2
- Issue #3533[1470] - Cmake version 3.5.1does not work (git ff26b35 2018-11-06)
- Issue #3526[1471] - Failed building hpx-1.2.0-rc1 on Ubuntu16.04 x86-64 Virtualbox VM
- Issue #3512[1472] - Build on aarch64 fails
- Issue #3475[1473] - HPX fails to link if the MPI parcelport is enabled
- Issue #3462[1474] - CMake configuration shows a minor and inconsequential failure to create a symlink
- Issue #3461[1475] - Compilation Problems with the most recent Clang
- Issue #3460[1476] - Deadlock when create_partitioner fails (assertion fails) in debug mode
- Issue #3455[1477] - HPX build failing with HWLOC errors on POWER8 with hwloc 1.8
- Issue #3438[1478] - HPX no longer builds on IBM POWER8
- Issue #3426[1479] - hpx build failed on MacOS
- Issue #3424[1480] - CircleCI builds broken for forked repositories
- Issue #3422[1481] - Benchmarks in tests.performance.local are not run nightly
- Issue #3408[1482] - CMake Targets for HPX
- Issue #3399[1483] - processing unit out of bounds
- Issue #3395[1484] - Floating point bug in hpx/runtime/threads/policies/scheduler_base.hpp
- Issue #3378[1485] - compile error with lcos::communicator
- Issue #3376[1486] - Failed to build HPX with APEX using clang
- Issue #3366[1487] - Adapted Safe_Object example fails for –hpx:threads > 1
- Issue #3360[1488] - Segmentation fault when passing component id as parameter
- Issue #3358[1489] - HPX runtime hangs after multiple (~thousands) start-stop sequences
- Issue #3352[1490] - Support TCP provider in libfabric ParcelPort

[1469] https://github.com/STEllAR-GROUP/hpx/issues/3538
[1470] https://github.com/STEllAR-GROUP/hpx/issues/3533
[1471] https://github.com/STEllAR-GROUP/hpx/issues/3526
[1472] https://github.com/STEllAR-GROUP/hpx/issues/3512
[1473] https://github.com/STEllAR-GROUP/hpx/issues/3475
[1474] https://github.com/STEllAR-GROUP/hpx/issues/3462
[1475] https://github.com/STEllAR-GROUP/hpx/issues/3461
[1476] https://github.com/STEllAR-GROUP/hpx/issues/3460
[1477] https://github.com/STEllAR-GROUP/hpx/issues/3455
[1478] https://github.com/STEllAR-GROUP/hpx/issues/3438
[1479] https://github.com/STEllAR-GROUP/hpx/issues/3426
[1480] https://github.com/STEllAR-GROUP/hpx/issues/3424
[1481] https://github.com/STEllAR-GROUP/hpx/issues/3422
[1482] https://github.com/STEllAR-GROUP/hpx/issues/3408
[1483] https://github.com/STEllAR-GROUP/hpx/issues/3399
[1484] https://github.com/STEllAR-GROUP/hpx/issues/3395
[1485] https://github.com/STEllAR-GROUP/hpx/issues/3378
[1486] https://github.com/STEllAR-GROUP/hpx/issues/3376
[1487] https://github.com/STEllAR-GROUP/hpx/issues/3366
[1488] https://github.com/STEllAR-GROUP/hpx/issues/3360
[1489] https://github.com/STEllAR-GROUP/hpx/issues/3358
[1490] https://github.com/STEllAR-GROUP/hpx/issues/3352

- Issue #3342[1491] - undefined reference to __atomic_load_16
- Issue #3339[1492] - setting command line options/flags from init cfg is not obvious
- Issue #3325[1493] - AGAS migrates components prematurely
- Issue #3321[1494] - hpx bad_parameter handling is awful
- Issue #3318[1495] - Benchmarks fail to build with C++11
- Issue #3304[1496] - hpx::threads::run_as_hpx_thread does not properly handle exceptions
- Issue #3300[1497] - Setting pu step or offset results in no threads in default pool
- Issue #3297[1498] - Crash with APEX when running Phylanx lra_csv with > 1 thread
- Issue #3296[1499] - Building HPX with APEX configuration gives compiler warnings
- Issue #3290[1500] - make tests failing at hello_world_component
- Issue #3285[1501] - possible compilation error when "using namespace std;" is defined before including "hpx" headers files
- Issue #3280[1502] - HPX fails on OSX
- Issue #3272[1503] - CircleCI does not upload generated docker image any more
- Issue #3270[1504] - Error when compiling CUDA examples
- Issue #3267[1505] - `tests.unit.host_.block_allocator` fails occasionally
- Issue #3264[1506] - Possible move to Sphinx for documentation
- Issue #3263[1507] - Documentation improvements
- Issue #3259[1508] - `set_parcel_write_handler` test fails occasionally
- Issue #3258[1509] - Links to source code in documentation are broken
- Issue #3247[1510] - Rare `tests.unit.host_.block_allocator` test failure on 1.1.0-rc1
- Issue #3244[1511] - Slowing down and speeding up an interval_timer
- Issue #3215[1512] - Cannot build both tests and examples on MSVC with pseudo-dependencies enabled
- Issue #3195[1513] - Unnecessary customization point route causing performance penalty

1491 https://github.com/STEllAR-GROUP/hpx/issues/3342
1492 https://github.com/STEllAR-GROUP/hpx/issues/3339
1493 https://github.com/STEllAR-GROUP/hpx/issues/3325
1494 https://github.com/STEllAR-GROUP/hpx/issues/3321
1495 https://github.com/STEllAR-GROUP/hpx/issues/3318
1496 https://github.com/STEllAR-GROUP/hpx/issues/3304
1497 https://github.com/STEllAR-GROUP/hpx/issues/3300
1498 https://github.com/STEllAR-GROUP/hpx/issues/3297
1499 https://github.com/STEllAR-GROUP/hpx/issues/3296
1500 https://github.com/STEllAR-GROUP/hpx/issues/3290
1501 https://github.com/STEllAR-GROUP/hpx/issues/3285
1502 https://github.com/STEllAR-GROUP/hpx/issues/3280
1503 https://github.com/STEllAR-GROUP/hpx/issues/3272
1504 https://github.com/STEllAR-GROUP/hpx/issues/3270
1505 https://github.com/STEllAR-GROUP/hpx/issues/3267
1506 https://github.com/STEllAR-GROUP/hpx/issues/3264
1507 https://github.com/STEllAR-GROUP/hpx/issues/3263
1508 https://github.com/STEllAR-GROUP/hpx/issues/3259
1509 https://github.com/STEllAR-GROUP/hpx/issues/3258
1510 https://github.com/STEllAR-GROUP/hpx/issues/3247
1511 https://github.com/STEllAR-GROUP/hpx/issues/3244
1512 https://github.com/STEllAR-GROUP/hpx/issues/3215
1513 https://github.com/STEllAR-GROUP/hpx/issues/3195

- Issue #3088[1514] - A strange thing in parallel::sort.
- Issue #2650[1515] - libfabric support for passive endpoints
- Issue #1205[1516] - TSS is broken

Closed pull requests

- PR #3542[1517] - Fix numa lookup from pu when using hwloc 2.x
- PR #3541[1518] - Fixing the build system of the MPI parcelport
- PR #3540[1519] - Updating HPX people section
- PR #3539[1520] - Splitting test to avoid OOM on CircleCI
- PR #3537[1521] - Fix guided exec
- PR #3536[1522] - Updating grants which support the LSU team
- PR #3535[1523] - Fix hiding of docker credentials
- PR #3534[1524] - Fixing #3533
- PR #3532[1525] - fixing minor doc typo –hpx:print-counter-at arg
- PR #3530[1526] - Changing APEX default tag to v2.1.0
- PR #3529[1527] - Remove leftover security options and documentation
- PR #3528[1528] - Fix hwloc version check
- PR #3524[1529] - Do not build guided pool examples with older GCC compilers
- PR #3523[1530] - Fix logging regression
- PR #3522[1531] - Fix more warnings
- PR #3521[1532] - Fixing argument handling in induction and reduction clauses for parallel::for_loop
- PR #3520[1533] - Remove docs symlink and versioned docs folders
- PR #3519[1534] - hpxMP release
- PR #3518[1535] - Change all steps to use new docker image on CircleCI

1514 https://github.com/STEllAR-GROUP/hpx/issues/3088
1515 https://github.com/STEllAR-GROUP/hpx/issues/2650
1516 https://github.com/STEllAR-GROUP/hpx/issues/1205
1517 https://github.com/STEllAR-GROUP/hpx/pull/3542
1518 https://github.com/STEllAR-GROUP/hpx/pull/3541
1519 https://github.com/STEllAR-GROUP/hpx/pull/3540
1520 https://github.com/STEllAR-GROUP/hpx/pull/3539
1521 https://github.com/STEllAR-GROUP/hpx/pull/3537
1522 https://github.com/STEllAR-GROUP/hpx/pull/3536
1523 https://github.com/STEllAR-GROUP/hpx/pull/3535
1524 https://github.com/STEllAR-GROUP/hpx/pull/3534
1525 https://github.com/STEllAR-GROUP/hpx/pull/3532
1526 https://github.com/STEllAR-GROUP/hpx/pull/3530
1527 https://github.com/STEllAR-GROUP/hpx/pull/3529
1528 https://github.com/STEllAR-GROUP/hpx/pull/3528
1529 https://github.com/STEllAR-GROUP/hpx/pull/3524
1530 https://github.com/STEllAR-GROUP/hpx/pull/3523
1531 https://github.com/STEllAR-GROUP/hpx/pull/3522
1532 https://github.com/STEllAR-GROUP/hpx/pull/3521
1533 https://github.com/STEllAR-GROUP/hpx/pull/3520
1534 https://github.com/STEllAR-GROUP/hpx/pull/3519
1535 https://github.com/STEllAR-GROUP/hpx/pull/3518

- PR #3516[1536] - Drop usage of deprecated facilities removed in C++17
- PR #3515[1537] - Remove remaining uses of Boost.TypeTraits
- PR #3513[1538] - Fixing a CMake problem when trying to use libfabric
- PR #3508[1539] - Remove memory_block component
- PR #3507[1540] - Propagating the MPI compile definitions to all relevant targets
- PR #3503[1541] - Update documentation colors and logo
- PR #3502[1542] - Fix bogus `throws` bindings in scheduled_thread_pool_impl
- PR #3501[1543] - Split parallel::remove_if tests to avoid OOM on CircleCI
- PR #3500[1544] - Support NONAMEPREFIX in add_hpx_library()
- PR #3497[1545] - Note that cuda support requires cmake 3.9
- PR #3495[1546] - Fixing dataflow
- PR #3493[1547] - Remove deprecated options for 1.2.0 part 2
- PR #3492[1548] - Add CUDA_LINK_LIBRARIES_KEYWORD to allow PRIVATE keyword in linkage t...
- PR #3491[1549] - Changing Base docker image
- PR #3490[1550] - Don't create tasks immediately with hpx::apply
- PR #3489[1551] - Remove deprecated options for 1.2.0
- PR #3488[1552] - Revert "Use BUILD_INTERFACE generator expression to fix cmake flag exports"
- PR #3487[1553] - Revert "Fixing type attribute warning for transfer_action"
- PR #3485[1554] - Use BUILD_INTERFACE generator expression to fix cmake flag exports
- PR #3483[1555] - Fixing type attribute warning for transfer_action
- PR #3481[1556] - Remove unused variables
- PR #3480[1557] - Towards a more lightweight transfer action
- PR #3479[1558] - Fix FLAGS - Use correct version of target_compile_options

1536 https://github.com/STEllAR-GROUP/hpx/pull/3516
1537 https://github.com/STEllAR-GROUP/hpx/pull/3515
1538 https://github.com/STEllAR-GROUP/hpx/pull/3513
1539 https://github.com/STEllAR-GROUP/hpx/pull/3508
1540 https://github.com/STEllAR-GROUP/hpx/pull/3507
1541 https://github.com/STEllAR-GROUP/hpx/pull/3503
1542 https://github.com/STEllAR-GROUP/hpx/pull/3502
1543 https://github.com/STEllAR-GROUP/hpx/pull/3501
1544 https://github.com/STEllAR-GROUP/hpx/pull/3500
1545 https://github.com/STEllAR-GROUP/hpx/pull/3497
1546 https://github.com/STEllAR-GROUP/hpx/pull/3495
1547 https://github.com/STEllAR-GROUP/hpx/pull/3493
1548 https://github.com/STEllAR-GROUP/hpx/pull/3492
1549 https://github.com/STEllAR-GROUP/hpx/pull/3491
1550 https://github.com/STEllAR-GROUP/hpx/pull/3490
1551 https://github.com/STEllAR-GROUP/hpx/pull/3489
1552 https://github.com/STEllAR-GROUP/hpx/pull/3488
1553 https://github.com/STEllAR-GROUP/hpx/pull/3487
1554 https://github.com/STEllAR-GROUP/hpx/pull/3485
1555 https://github.com/STEllAR-GROUP/hpx/pull/3483
1556 https://github.com/STEllAR-GROUP/hpx/pull/3481
1557 https://github.com/STEllAR-GROUP/hpx/pull/3480
1558 https://github.com/STEllAR-GROUP/hpx/pull/3479

- PR #3478[1559] - Making sure the application's exit code is properly propagated back to the OS
- PR #3476[1560] - Don't print docker credentials as part of the environment.
- PR #3473[1561] - Fixing invalid cmake code if no jemalloc prefix was given
- PR #3472[1562] - Attempting to work around recent clang test compilation failures
- PR #3471[1563] - Enable jemalloc on windows
- PR #3470[1564] - Updates readme
- PR #3468[1565] - Avoid hang if there is an exception thrown during startup
- PR #3467[1566] - Add compiler specific fallthrough attributes if C++17 attribute is not available
- PR #3466[1567] - - bugfix : fix compilation with llvm-7.0
- PR #3465[1568] - This patch adds various optimizations extracted from the thread_local_allocator work
- PR #3464[1569] - Check for forked repos in CircleCI docker push step
- PR #3463[1570] - - cmake : create the parent directory before symlinking
- PR #3459[1571] - Remove unused/incomplete functionality from util/logging
- PR #3458[1572] - Fix a problem with scope of CMAKE_CXX_FLAGS and hpx_add_compile_flag
- PR #3457[1573] - Fixing more size_t -> int16_t (and similar) warnings
- PR #3456[1574] - Add #ifdefs to topology.cpp to support old hwloc versions again
- PR #3454[1575] - Fixing warnings related to silent conversion of size_t –> int16_t
- PR #3451[1576] - Add examples as unit tests
- PR #3450[1577] - Constexpr-fying bind and other functional facilities
- PR #3446[1578] - Fix some thread suspension timeouts
- PR #3445[1579] - Fix various warnings
- PR #3443[1580] - Only enable service pool config options if pools are enabled
- PR #3441[1581] - Fix missing closing brackets in documentation

[1559] https://github.com/STEllAR-GROUP/hpx/pull/3478
[1560] https://github.com/STEllAR-GROUP/hpx/pull/3476
[1561] https://github.com/STEllAR-GROUP/hpx/pull/3473
[1562] https://github.com/STEllAR-GROUP/hpx/pull/3472
[1563] https://github.com/STEllAR-GROUP/hpx/pull/3471
[1564] https://github.com/STEllAR-GROUP/hpx/pull/3470
[1565] https://github.com/STEllAR-GROUP/hpx/pull/3468
[1566] https://github.com/STEllAR-GROUP/hpx/pull/3467
[1567] https://github.com/STEllAR-GROUP/hpx/pull/3466
[1568] https://github.com/STEllAR-GROUP/hpx/pull/3465
[1569] https://github.com/STEllAR-GROUP/hpx/pull/3464
[1570] https://github.com/STEllAR-GROUP/hpx/pull/3463
[1571] https://github.com/STEllAR-GROUP/hpx/pull/3459
[1572] https://github.com/STEllAR-GROUP/hpx/pull/3458
[1573] https://github.com/STEllAR-GROUP/hpx/pull/3457
[1574] https://github.com/STEllAR-GROUP/hpx/pull/3456
[1575] https://github.com/STEllAR-GROUP/hpx/pull/3454
[1576] https://github.com/STEllAR-GROUP/hpx/pull/3451
[1577] https://github.com/STEllAR-GROUP/hpx/pull/3450
[1578] https://github.com/STEllAR-GROUP/hpx/pull/3446
[1579] https://github.com/STEllAR-GROUP/hpx/pull/3445
[1580] https://github.com/STEllAR-GROUP/hpx/pull/3443
[1581] https://github.com/STEllAR-GROUP/hpx/pull/3441

- PR #3439[1582] - Use correct MPI CXX libraries for MPI parcelport
- PR #3436[1583] - Add projection function to find_* (and fix very bad bug)
- PR #3435[1584] - Fixing 1205
- PR #3434[1585] - Fix threads cores
- PR #3433[1586] - Add Heise Online to release announcement list
- PR #3432[1587] - Don't track task dependencies for distributed runs
- PR #3431[1588] - Circle CI setting changes for hpxMP
- PR #3430[1589] - Fix unused params warning
- PR #3429[1590] - One thread per core
- PR #3428[1591] - This suppresses a deprecation warning that is being issued by MSVC 19.15.26726
- PR #3427[1592] - Fixes #3426
- PR #3425[1593] - Use source cache and workspace between job steps on CircleCI
- PR #3421[1594] - Add CDash timing output to future overhead test (for graphs)
- PR #3420[1595] - Add guided_pool_executor
- PR #3419[1596] - Fix typo in CircleCI config
- PR #3418[1597] - Add sphinx documentation
- PR #3415[1598] - Scheduler NUMA hint and shared priority scheduler
- PR #3414[1599] - Adding step to synchronize the APEX release
- PR #3413[1600] - Fixing multiple defines of APEX_HAVE_HPX
- PR #3412[1601] - Fixes linking with libhpx_wrap error with BSD and Windows based systems
- PR #3410[1602] - Fix typo in CMakeLists.txt
- PR #3409[1603] - Fix brackets and indentation in existing_performance_counters.qbk
- PR #3407[1604] - Fix unused param and extra ; warnings emitted by gcc 8.x

[1582] https://github.com/STEllAR-GROUP/hpx/pull/3439
[1583] https://github.com/STEllAR-GROUP/hpx/pull/3436
[1584] https://github.com/STEllAR-GROUP/hpx/pull/3435
[1585] https://github.com/STEllAR-GROUP/hpx/pull/3434
[1586] https://github.com/STEllAR-GROUP/hpx/pull/3433
[1587] https://github.com/STEllAR-GROUP/hpx/pull/3432
[1588] https://github.com/STEllAR-GROUP/hpx/pull/3431
[1589] https://github.com/STEllAR-GROUP/hpx/pull/3430
[1590] https://github.com/STEllAR-GROUP/hpx/pull/3429
[1591] https://github.com/STEllAR-GROUP/hpx/pull/3428
[1592] https://github.com/STEllAR-GROUP/hpx/pull/3427
[1593] https://github.com/STEllAR-GROUP/hpx/pull/3425
[1594] https://github.com/STEllAR-GROUP/hpx/pull/3421
[1595] https://github.com/STEllAR-GROUP/hpx/pull/3420
[1596] https://github.com/STEllAR-GROUP/hpx/pull/3419
[1597] https://github.com/STEllAR-GROUP/hpx/pull/3418
[1598] https://github.com/STEllAR-GROUP/hpx/pull/3415
[1599] https://github.com/STEllAR-GROUP/hpx/pull/3414
[1600] https://github.com/STEllAR-GROUP/hpx/pull/3413
[1601] https://github.com/STEllAR-GROUP/hpx/pull/3412
[1602] https://github.com/STEllAR-GROUP/hpx/pull/3410
[1603] https://github.com/STEllAR-GROUP/hpx/pull/3409
[1604] https://github.com/STEllAR-GROUP/hpx/pull/3407

- PR #3406[1605] - Adding thread local allocator and use it for future shared states
- PR #3405[1606] - Adding DHPX_HAVE_THREAD_LOCAL_STORAGE=ON to builds
- PR #3404[1607] - fixing multiple definition of main() in linux
- PR #3402[1608] - Allow debug option to be enabled only for Linux systems with dynamic main on
- PR #3401[1609] - Fix cuda_future_helper.h when compiling with C++11
- PR #3400[1610] - Fix floating point exception scheduler_base idle backoff
- PR #3398[1611] - Atomic future state
- PR #3397[1612] - Fixing code for older gcc versions
- PR #3396[1613] - Allowing to register thread event functions (start/stop/error)
- PR #3394[1614] - Fix small mistake in primary_namespace_server.cpp
- PR #3393[1615] - Explicitly instantiate configured schedulers
- PR #3392[1616] - Add performance counters background overhead and background work duration
- PR #3391[1617] - Adapt integration of HPXMP to latest build system changes
- PR #3390[1618] - Make AGAS measurements optional
- PR #3389[1619] - Fix deadlock during shutdown
- PR #3388[1620] - Add several functionalities allowing to optimize synchronous action invocation
- PR #3387[1621] - Add cmake option to opt out of fail-compile tests
- PR #3386[1622] - Adding support for boost::container::small_vector to dataflow
- PR #3385[1623] - Adds Debug option for hpx initializing from main
- PR #3384[1624] - This hopefully fixes two tests that occasionally fail
- PR #3383[1625] - Making sure thread local storage is enable for hpxMP
- PR #3382[1626] - Fix usage of HPX_CAPTURE together with default value capture [=]
- PR #3381[1627] - Replace undefined instantiations of uniform_int_distribution

1605 https://github.com/STEllAR-GROUP/hpx/pull/3406
1606 https://github.com/STEllAR-GROUP/hpx/pull/3405
1607 https://github.com/STEllAR-GROUP/hpx/pull/3404
1608 https://github.com/STEllAR-GROUP/hpx/pull/3402
1609 https://github.com/STEllAR-GROUP/hpx/pull/3401
1610 https://github.com/STEllAR-GROUP/hpx/pull/3400
1611 https://github.com/STEllAR-GROUP/hpx/pull/3398
1612 https://github.com/STEllAR-GROUP/hpx/pull/3397
1613 https://github.com/STEllAR-GROUP/hpx/pull/3396
1614 https://github.com/STEllAR-GROUP/hpx/pull/3394
1615 https://github.com/STEllAR-GROUP/hpx/pull/3393
1616 https://github.com/STEllAR-GROUP/hpx/pull/3392
1617 https://github.com/STEllAR-GROUP/hpx/pull/3391
1618 https://github.com/STEllAR-GROUP/hpx/pull/3390
1619 https://github.com/STEllAR-GROUP/hpx/pull/3389
1620 https://github.com/STEllAR-GROUP/hpx/pull/3388
1621 https://github.com/STEllAR-GROUP/hpx/pull/3387
1622 https://github.com/STEllAR-GROUP/hpx/pull/3386
1623 https://github.com/STEllAR-GROUP/hpx/pull/3385
1624 https://github.com/STEllAR-GROUP/hpx/pull/3384
1625 https://github.com/STEllAR-GROUP/hpx/pull/3383
1626 https://github.com/STEllAR-GROUP/hpx/pull/3382
1627 https://github.com/STEllAR-GROUP/hpx/pull/3381

- PR #3380[1628] - Add missing semicolons to uses of HPX_COMPILER_FENCE
- PR #3379[1629] - Fixing #3378
- PR #3377[1630] - Adding build system support to integrate hpxmp into hpx at the user's machine
- PR #3375[1631] - Replacing wrapper for __libc_start_main with main
- PR #3374[1632] - Adds hpx_wrap to HPX_LINK_LIBRARIES which links only when specified.
- PR #3373[1633] - Forcing cache settings in HPXConfig.cmake to guarantee updated values
- PR #3372[1634] - Fix some more c++11 build problems
- PR #3371[1635] - Adds HPX_LINKER_FLAGS to HPX applications without editing their source codes
- PR #3370[1636] - util::format: add type_specifier<> specializations for %!s(MISSING) and %!l(MISSING)s
- PR #3369[1637] - Adding configuration option to allow explicit disable of the new hpx_main feature on Linux
- PR #3368[1638] - Updates doc with recent hpx_wrap implementation
- PR #3367[1639] - Adds Mac OS implementation to hpx_main.hpp
- PR #3365[1640] - Fix order of hpx libs in HPX_CONF_LIBRARIES.
- PR #3363[1641] - Apex fixing null wrapper
- PR #3361[1642] - Making sure all parcels get destroyed on an HPX thread (TCP pp)
- PR #3359[1643] - Feature/improveerrorforcompiler
- PR #3357[1644] - Static/dynamic executable implementation
- PR #3355[1645] - Reverting changes introduced by #3283 as those make applications hang
- PR #3354[1646] - Add external dependencies to HPX_LIBRARY_DIR
- PR #3353[1647] - Fix libfabric tcp
- PR #3351[1648] - Move obsolete header to tests directory.
- PR #3350[1649] - Renaming two functions to avoid problem described in #3285
- PR #3349[1650] - Make idle backoff exponential with maximum sleep time

[1628] https://github.com/STEllAR-GROUP/hpx/pull/3380
[1629] https://github.com/STEllAR-GROUP/hpx/pull/3379
[1630] https://github.com/STEllAR-GROUP/hpx/pull/3377
[1631] https://github.com/STEllAR-GROUP/hpx/pull/3375
[1632] https://github.com/STEllAR-GROUP/hpx/pull/3374
[1633] https://github.com/STEllAR-GROUP/hpx/pull/3373
[1634] https://github.com/STEllAR-GROUP/hpx/pull/3372
[1635] https://github.com/STEllAR-GROUP/hpx/pull/3371
[1636] https://github.com/STEllAR-GROUP/hpx/pull/3370
[1637] https://github.com/STEllAR-GROUP/hpx/pull/3369
[1638] https://github.com/STEllAR-GROUP/hpx/pull/3368
[1639] https://github.com/STEllAR-GROUP/hpx/pull/3367
[1640] https://github.com/STEllAR-GROUP/hpx/pull/3365
[1641] https://github.com/STEllAR-GROUP/hpx/pull/3363
[1642] https://github.com/STEllAR-GROUP/hpx/pull/3361
[1643] https://github.com/STEllAR-GROUP/hpx/pull/3359
[1644] https://github.com/STEllAR-GROUP/hpx/pull/3357
[1645] https://github.com/STEllAR-GROUP/hpx/pull/3355
[1646] https://github.com/STEllAR-GROUP/hpx/pull/3354
[1647] https://github.com/STEllAR-GROUP/hpx/pull/3353
[1648] https://github.com/STEllAR-GROUP/hpx/pull/3351
[1649] https://github.com/STEllAR-GROUP/hpx/pull/3350
[1650] https://github.com/STEllAR-GROUP/hpx/pull/3349

- PR #3347[1651] - Replace *simple_component** with *component** in the Documentation
- PR #3346[1652] - Fix CMakeLists.txt example in quick start
- PR #3345[1653] - Fix automatic setting of HPX_MORE_THAN_64_THREADS
- PR #3344[1654] - Reduce amount of information printed for unknown command line options
- PR #3343[1655] - Safeguard HPX against destruction in global contexts
- PR #3341[1656] - Allowing for all command line options to be used as configuration settings
- PR #3340[1657] - Always convert inspect results to JUnit XML
- PR #3336[1658] - Only run docker push on master on CircleCI
- PR #3335[1659] - Update description of hpx.os_threads config parameter.
- PR #3334[1660] - Making sure early logging settings don't get mixed with others
- PR #3333[1661] - Update CMake links and versions in documentation
- PR #3332[1662] - Add notes on target suffixes to CMake documentation
- PR #3331[1663] - Add quickstart section to documentation
- PR #3330[1664] - Rename resource_partitioner test to avoid conflicts with pseudodependencies
- PR #3328[1665] - Making sure object is pinned while executing actions, even if action returns a future
- PR #3327[1666] - Add missing std::forward to tuple.hpp
- PR #3326[1667] - Make sure logging is up and running while modules are being discovered.
- PR #3324[1668] - Replace C++14 overload of std::equal with C++11 code.
- PR #3323[1669] - Fix a missing apex thread data (wrapper) initialization
- PR #3320[1670] - Adding support for -std=c++2a (define *HPX_WITH_CXX2A=On*)
- PR #3319[1671] - Replacing C++14 feature with equivalent C++11 code
- PR #3317[1672] - Fix compilation with VS 15.7.1 and /std:c++latest
- PR #3316[1673] - Fix includes for 1d_stencil_*_omp examples

1651 https://github.com/STEllAR-GROUP/hpx/pull/3347
1652 https://github.com/STEllAR-GROUP/hpx/pull/3346
1653 https://github.com/STEllAR-GROUP/hpx/pull/3345
1654 https://github.com/STEllAR-GROUP/hpx/pull/3344
1655 https://github.com/STEllAR-GROUP/hpx/pull/3343
1656 https://github.com/STEllAR-GROUP/hpx/pull/3341
1657 https://github.com/STEllAR-GROUP/hpx/pull/3340
1658 https://github.com/STEllAR-GROUP/hpx/pull/3336
1659 https://github.com/STEllAR-GROUP/hpx/pull/3335
1660 https://github.com/STEllAR-GROUP/hpx/pull/3334
1661 https://github.com/STEllAR-GROUP/hpx/pull/3333
1662 https://github.com/STEllAR-GROUP/hpx/pull/3332
1663 https://github.com/STEllAR-GROUP/hpx/pull/3331
1664 https://github.com/STEllAR-GROUP/hpx/pull/3330
1665 https://github.com/STEllAR-GROUP/hpx/pull/3328
1666 https://github.com/STEllAR-GROUP/hpx/pull/3327
1667 https://github.com/STEllAR-GROUP/hpx/pull/3326
1668 https://github.com/STEllAR-GROUP/hpx/pull/3324
1669 https://github.com/STEllAR-GROUP/hpx/pull/3323
1670 https://github.com/STEllAR-GROUP/hpx/pull/3320
1671 https://github.com/STEllAR-GROUP/hpx/pull/3319
1672 https://github.com/STEllAR-GROUP/hpx/pull/3317
1673 https://github.com/STEllAR-GROUP/hpx/pull/3316

- PR #3314[1674] - Remove some unused parameter warnings
- PR #3313[1675] - Fix pu-step and pu-offset command line options
- PR #3312[1676] - Add conversion of inspect reports to JUnit XML
- PR #3311[1677] - Fix escaping of closing braces in format specification syntax
- PR #3310[1678] - Don't overwrite user settings with defaults in registration database
- PR #3309[1679] - Fixing potential stack overflow for dataflow
- PR #3308[1680] - This updates the .clang-format configuration file to utilize newer features
- PR #3306[1681] - Marking migratable objects in their gid to allow not handling migration in AGAS
- PR #3305[1682] - Add proper exception handling to run_as_hpx_thread
- PR #3303[1683] - Changed std::rand to a better inbuilt PRNG Generator
- PR #3302[1684] - All non-migratable (simple) components now encode their lva and component type in their gid
- PR #3301[1685] - Add nullptr_t overloads to resource partitioner
- PR #3298[1686] - Apex task wrapper memory bug
- PR #3295[1687] - Fix mistakes after merge of CircleCI config
- PR #3294[1688] - Fix partitioned vector include in partitioned_vector_find tests
- PR #3293[1689] - Adding emplace support to promise and make_ready_future
- PR #3292[1690] - Add new cuda kernel synchronization with hpx::future demo
- PR #3291[1691] - Fixes #3290
- PR #3289[1692] - Fixing Docker image creation
- PR #3288[1693] - Avoid allocating shared state for wait_all
- PR #3287[1694] - Fixing /scheduler/utilization/instantaneous performance counter
- PR #3286[1695] - dataflow() and future::then() use sync policy where possible
- PR #3284[1696] - Background thread can use relaxed atomics to manipulate thread state

1674 https://github.com/STEllAR-GROUP/hpx/pull/3314
1675 https://github.com/STEllAR-GROUP/hpx/pull/3313
1676 https://github.com/STEllAR-GROUP/hpx/pull/3312
1677 https://github.com/STEllAR-GROUP/hpx/pull/3311
1678 https://github.com/STEllAR-GROUP/hpx/pull/3310
1679 https://github.com/STEllAR-GROUP/hpx/pull/3309
1680 https://github.com/STEllAR-GROUP/hpx/pull/3308
1681 https://github.com/STEllAR-GROUP/hpx/pull/3306
1682 https://github.com/STEllAR-GROUP/hpx/pull/3305
1683 https://github.com/STEllAR-GROUP/hpx/pull/3303
1684 https://github.com/STEllAR-GROUP/hpx/pull/3302
1685 https://github.com/STEllAR-GROUP/hpx/pull/3301
1686 https://github.com/STEllAR-GROUP/hpx/pull/3298
1687 https://github.com/STEllAR-GROUP/hpx/pull/3295
1688 https://github.com/STEllAR-GROUP/hpx/pull/3294
1689 https://github.com/STEllAR-GROUP/hpx/pull/3293
1690 https://github.com/STEllAR-GROUP/hpx/pull/3292
1691 https://github.com/STEllAR-GROUP/hpx/pull/3291
1692 https://github.com/STEllAR-GROUP/hpx/pull/3289
1693 https://github.com/STEllAR-GROUP/hpx/pull/3288
1694 https://github.com/STEllAR-GROUP/hpx/pull/3287
1695 https://github.com/STEllAR-GROUP/hpx/pull/3286
1696 https://github.com/STEllAR-GROUP/hpx/pull/3284

- PR #3283[1697] - Do not unwrap ready future
- PR #3282[1698] - Fix virtual method override warnings in static schedulers
- PR #3281[1699] - Disable set_area_membind_nodeset for OSX
- PR #3279[1700] - Add two variations to the future_overhead benchmark
- PR #3278[1701] - Fix circleci workspace
- PR #3277[1702] - Support external plugins
- PR #3276[1703] - Fix missing parenthesis in hello_compute.cu.
- PR #3274[1704] - Reinit counters synchronously in reinit_counters test
- PR #3273[1705] - Splitting tests to avoid compiler OOM
- PR #3271[1706] - Remove leftover code from context_generic_context.hpp
- PR #3269[1707] - Fix bulk_construct with count = 0
- PR #3268[1708] - Replace constexpr with HPX_CXX14_CONSTEXPR and HPX_CONSTEXPR
- PR #3266[1709] - Replace boost::format with custom sprintf-based implementation
- PR #3265[1710] - Split parallel tests on CircleCI
- PR #3262[1711] - Making sure documentation correctly links to source files
- PR #3261[1712] - Apex refactoring fix rebind
- PR #3260[1713] - Isolate performance counter parser into a separate TU
- PR #3256[1714] - Post 1.1.0 version bumps
- PR #3254[1715] - Adding trait for actions allowing to make runtime decision on whether to execute it directly
- PR #3253[1716] - Bump minimal supported Boost to 1.58.0
- PR #3251[1717] - Adds new feature: changing interval used in interval_timer (issue 3244)
- PR #3239[1718] - Changing std::rand() to a better inbuilt PRNG generator.
- PR #3234[1719] - Disable background thread when networking is off

1697 https://github.com/STEllAR-GROUP/hpx/pull/3283
1698 https://github.com/STEllAR-GROUP/hpx/pull/3282
1699 https://github.com/STEllAR-GROUP/hpx/pull/3281
1700 https://github.com/STEllAR-GROUP/hpx/pull/3279
1701 https://github.com/STEllAR-GROUP/hpx/pull/3278
1702 https://github.com/STEllAR-GROUP/hpx/pull/3277
1703 https://github.com/STEllAR-GROUP/hpx/pull/3276
1704 https://github.com/STEllAR-GROUP/hpx/pull/3274
1705 https://github.com/STEllAR-GROUP/hpx/pull/3273
1706 https://github.com/STEllAR-GROUP/hpx/pull/3271
1707 https://github.com/STEllAR-GROUP/hpx/pull/3269
1708 https://github.com/STEllAR-GROUP/hpx/pull/3268
1709 https://github.com/STEllAR-GROUP/hpx/pull/3266
1710 https://github.com/STEllAR-GROUP/hpx/pull/3265
1711 https://github.com/STEllAR-GROUP/hpx/pull/3262
1712 https://github.com/STEllAR-GROUP/hpx/pull/3261
1713 https://github.com/STEllAR-GROUP/hpx/pull/3260
1714 https://github.com/STEllAR-GROUP/hpx/pull/3256
1715 https://github.com/STEllAR-GROUP/hpx/pull/3254
1716 https://github.com/STEllAR-GROUP/hpx/pull/3253
1717 https://github.com/STEllAR-GROUP/hpx/pull/3251
1718 https://github.com/STEllAR-GROUP/hpx/pull/3239
1719 https://github.com/STEllAR-GROUP/hpx/pull/3234

- PR #3232[1720] - Clean up suspension tests
- PR #3230[1721] - Add optional scheduler mode parameter to create_thread_pool function
- PR #3228[1722] - Allow suspension also on static schedulers
- PR #3163[1723] - libfabric parcelport w/o HPX_PARCELPORT_LIBFABRIC_ENDPOINT_RDM
- PR #3036[1724] - Switching to CircleCI 2.0

2.11.8 *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[1725] and/or MPI[1726].
- 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[1727]. We have also continued to implement these algorithms for the distributed use case (for segmented data structures, such as `hpx::partitioned_vector`).
- Added a compatibility layer for `std::thread`, `std::mutex`, and `std::condition_variable` allowing for the code to use those facilities where available and to fall back to the corresponding Boost facilities otherwise. The CMake[1728] 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[1729].
- 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.

[1720] https://github.com/STEllAR-GROUP/hpx/pull/3232
[1721] https://github.com/STEllAR-GROUP/hpx/pull/3230
[1722] https://github.com/STEllAR-GROUP/hpx/pull/3228
[1723] https://github.com/STEllAR-GROUP/hpx/pull/3163
[1724] https://github.com/STEllAR-GROUP/hpx/pull/3036
[1725] https://openmp.org/wp/
[1726] https://en.wikipedia.org/wiki/Message_Passing_Interface
[1727] http://www.open-std.org/jtc1/sc22/wg21
[1728] https://www.cmake.org
[1729] https://www.cmake.org

- The parallel algorithms adopted for C++17 restrict the iterator categories usable with those to at least forward iterators. Our implementation of the parallel algorithms was supporting input iterators (and output iterators) as well by simply falling back to sequential execution. We have now made our implementations conforming by requiring at least forward iterators. In order to enable the old behavior use the compatibility option `-DHPX_WITH_ALGORITHM_INPUT_ITERATOR_SUPPORT=On` on the CMake[1730] command line.
- We have added the functionalities allowing for LCOs being implemented using (simple) components. Before LCOs had to always be implemented using managed components.
- User defined components don't have to be default-constructible anymore. Return types from actions don't have to be default-constructible anymore either. Our serialization layer now in general supports non-default-constructible types.
- We have added a new launch policy `hpx::launch::lazy` that allows oneto defer the decision on what launch policy to use to the point of execution. This policy is initialized with a function (object) that – when invoked – is expected to produce the desired launch policy.

Breaking changes

- We have dropped support for the gcc compiler version V4.8. The minimal gcc version we now test on is gcc V4.9. The minimally required version of CMake[1731] is now V3.3.2.
- We have dropped support for the Visual Studio 2013 compiler version. The minimal Visual Studio version we now test on is Visual Studio 2015.5.
- We have dropped support for the Boost V1.51-V1.54. The minimal version of Boost we now test is Boost V1.55.
- We have dropped support for the `hpx::util::unwrapped` API. `hpx::util::unwrapped` will stay functional to some degree, until it finally gets removed in a later version of HPX. The functional usage of `hpx::util::unwrapped` should be changed to the new `hpx::util::unwrapping` function whereas the immediate usage should be replaced to `hpx::util::unwrap`.
- The performance counter names referring to properties as exposed by the threading subsystem have changes as those now additionally have to specify the thread-pool. See the corresponding documentation for more details.
- The overloads of `hpx::async` that invoke an action do not perform implicit unwrapping of the returned future anymore in case the invoked function does return a future in the first place. In this case `hpx::async` now returns a `hpx::future<future<T>>` making its behavior conforming to its local counterpart.
- We have replaced the use of `boost::exception_ptr` in our APIs with the equivalent `std::exception_ptr`. Please change your codes accordingly. No compatibility settings are provided.
- We have removed the compatibility settings for `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` and `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY` as their life-cycle has reached its end.
- We have removed the experimental thread schedulers hierarchy_scheduler, periodic_priority_scheduler and throttling_scheduler in an effort to clean up and consolidate our thread schedulers.

[1730] https://www.cmake.org
[1731] https://www.cmake.org

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #3250[1732] - Apex refactoring with guids
- PR #3249[1733] - Updating People.qbk
- PR #3246[1734] - Assorted fixes for CUDA
- PR #3245[1735] - Apex refactoring with guids
- PR #3242[1736] - Modify task counting in thread_queue.hpp
- PR #3240[1737] - Fixed typos
- PR #3238[1738] - Readding accidentally removed std::abort
- PR #3237[1739] - Adding Pipeline example
- PR #3236[1740] - Fixing memory_block
- PR #3233[1741] - Make schedule_thread take suspended threads into account
- Issue #3226[1742] - memory_block is breaking, signaling SIGSEGV on a thread on creation and freeing
- PR #3225[1743] - Applying quick fix for hwloc-2.0
- Issue #3224[1744] - HPX counters crashing the application
- PR #3223[1745] - Fix returns when setting config entries
- Issue #3222[1746] - Errors linking libhpx.so
- Issue #3221[1747] - HPX on Mac OS X with HWLoc 2.0.0 fails to run
- PR #3216[1748] - Reorder a variadic array to satisfy VS 2017 15.6
- PR #3214[1749] - Changed prerequisites.qbk to avoid confusion while building boost
- PR #3213[1750] - Relax locks for thread suspension to avoid holding locks when yielding
- PR #3212[1751] - Fix check in sequenced_executor test
- PR #3211[1752] - Use preinit_array to set argc/argv in init_globally example

[1732] https://github.com/STEllAR-GROUP/hpx/pull/3250
[1733] https://github.com/STEllAR-GROUP/hpx/pull/3249
[1734] https://github.com/STEllAR-GROUP/hpx/pull/3246
[1735] https://github.com/STEllAR-GROUP/hpx/pull/3245
[1736] https://github.com/STEllAR-GROUP/hpx/pull/3242
[1737] https://github.com/STEllAR-GROUP/hpx/pull/3240
[1738] https://github.com/STEllAR-GROUP/hpx/pull/3238
[1739] https://github.com/STEllAR-GROUP/hpx/pull/3237
[1740] https://github.com/STEllAR-GROUP/hpx/pull/3236
[1741] https://github.com/STEllAR-GROUP/hpx/pull/3233
[1742] https://github.com/STEllAR-GROUP/hpx/issues/3226
[1743] https://github.com/STEllAR-GROUP/hpx/pull/3225
[1744] https://github.com/STEllAR-GROUP/hpx/issues/3224
[1745] https://github.com/STEllAR-GROUP/hpx/pull/3223
[1746] https://github.com/STEllAR-GROUP/hpx/issues/3222
[1747] https://github.com/STEllAR-GROUP/hpx/issues/3221
[1748] https://github.com/STEllAR-GROUP/hpx/pull/3216
[1749] https://github.com/STEllAR-GROUP/hpx/pull/3214
[1750] https://github.com/STEllAR-GROUP/hpx/pull/3213
[1751] https://github.com/STEllAR-GROUP/hpx/pull/3212
[1752] https://github.com/STEllAR-GROUP/hpx/pull/3211

- PR #3210[1753] - Adapted parallel::{search | search_n} for Ranges TS (see #1668)
- PR #3209[1754] - Fix locking problems during shutdown
- Issue #3208[1755] - init_globally throwing a run-time error
- PR #3206[1756] - Addition of new arithmetic performance counter "Count"
- PR #3205[1757] - Fixing return type calculation for bulk_then_execute
- PR #3204[1758] - Changing std::rand() to a better inbuilt PRNG generator
- PR #3203[1759] - Resolving problems during shutdown for VS2015
- PR #3202[1760] - Making sure resource partitioner is not accessed if its not valid
- PR #3201[1761] - Fixing optional::swap
- Issue #3200[1762] - hpx::util::optional fails
- PR #3199[1763] - Fix sliding_semaphore test
- PR #3198[1764] - Set pre_main status before launching run_helper
- PR #3197[1765] - Update README.rst
- PR #3194[1766] - parallel::{fill|fill_n} updated for Ranges TS
- PR #3193[1767] - Updating Runtime.cpp by adding correct description of Performance counters during register
- PR #3191[1768] - Fix sliding_semaphore_2338 test
- PR #3190[1769] - Topology improvements
- PR #3189[1770] - Deleting one include of median from BOOST library to arithmetics_counter file
- PR #3188[1771] - Optionally disable printing of diagnostics during terminate
- PR #3187[1772] - Suppressing cmake warning issued by cmake > V3.11
- PR #3185[1773] - Remove unused scoped_unlock, unlock_guard_try
- PR #3184[1774] - Fix nqueen example
- PR #3183[1775] - Add runtime start/stop, resume/suspend and OpenMP benchmarks

1753 https://github.com/STEllAR-GROUP/hpx/pull/3210
1754 https://github.com/STEllAR-GROUP/hpx/pull/3209
1755 https://github.com/STEllAR-GROUP/hpx/issues/3208
1756 https://github.com/STEllAR-GROUP/hpx/pull/3206
1757 https://github.com/STEllAR-GROUP/hpx/pull/3205
1758 https://github.com/STEllAR-GROUP/hpx/pull/3204
1759 https://github.com/STEllAR-GROUP/hpx/pull/3203
1760 https://github.com/STEllAR-GROUP/hpx/pull/3202
1761 https://github.com/STEllAR-GROUP/hpx/pull/3201
1762 https://github.com/STEllAR-GROUP/hpx/issues/3200
1763 https://github.com/STEllAR-GROUP/hpx/pull/3199
1764 https://github.com/STEllAR-GROUP/hpx/pull/3198
1765 https://github.com/STEllAR-GROUP/hpx/pull/3197
1766 https://github.com/STEllAR-GROUP/hpx/pull/3194
1767 https://github.com/STEllAR-GROUP/hpx/pull/3193
1768 https://github.com/STEllAR-GROUP/hpx/pull/3191
1769 https://github.com/STEllAR-GROUP/hpx/pull/3190
1770 https://github.com/STEllAR-GROUP/hpx/pull/3189
1771 https://github.com/STEllAR-GROUP/hpx/pull/3188
1772 https://github.com/STEllAR-GROUP/hpx/pull/3187
1773 https://github.com/STEllAR-GROUP/hpx/pull/3185
1774 https://github.com/STEllAR-GROUP/hpx/pull/3184
1775 https://github.com/STEllAR-GROUP/hpx/pull/3183

- Issue #3182[1776] - bulk_then_execute has unexpected return type/does not compile
- Issue #3181[1777] - hwloc 2.0 breaks topo class and cannot be used
- Issue #3180[1778] - Schedulers that don't support suspend/resume are unusable
- PR #3179[1779] - Various minor changes to support FLeCSI
- PR #3178[1780] - Fix #3124
- PR #3177[1781] - Removed allgather
- PR #3176[1782] - Fixed Documentation for "using_hpx_pkgconfig"
- PR #3174[1783] - Add hpx::iostreams::ostream overload to format_to
- PR #3172[1784] - Fix lifo queue backend
- PR #3171[1785] - adding the missing unset() function to cpu_mask() for case of more than 64 threads
- PR #3170[1786] - Add cmake flag -DHPX_WITH_FAULT_TOLERANCE=ON (OFF by default)
- PR #3169[1787] - Adapted parallel::{count|count_if} for Ranges TS (see #1668)
- PR #3168[1788] - Changing used namespace for seq execution policy
- Issue #3167[1789] - Update GSoC projects
- Issue #3166[1790] - Application (Octotiger) gets stuck on hpx::finalize when only using one thread
- Issue #3165[1791] - Compilation of parallel algorithms with HPX_WITH_DATAPAR is broken
- PR #3164[1792] - Fixing component migration
- PR #3162[1793] - regex_from_pattern: escape regex special characters to avoid misinterpretation
- Issue #3161[1794] - Building HPX with hwloc 2.0.0 fails
- PR #3160[1795] - Fixing the handling of quoted command line arguments.
- PR #3158[1796] - Fixing a race with timed suspension (second attempt)
- PR #3157[1797] - Revert "Fixing a race with timed suspension"
- PR #3156[1798] - Fixing serialization of classes with incompatible serialize signature

[1776] https://github.com/STEllAR-GROUP/hpx/issues/3182
[1777] https://github.com/STEllAR-GROUP/hpx/issues/3181
[1778] https://github.com/STEllAR-GROUP/hpx/issues/3180
[1779] https://github.com/STEllAR-GROUP/hpx/pull/3179
[1780] https://github.com/STEllAR-GROUP/hpx/pull/3178
[1781] https://github.com/STEllAR-GROUP/hpx/pull/3177
[1782] https://github.com/STEllAR-GROUP/hpx/pull/3176
[1783] https://github.com/STEllAR-GROUP/hpx/pull/3174
[1784] https://github.com/STEllAR-GROUP/hpx/pull/3172
[1785] https://github.com/STEllAR-GROUP/hpx/pull/3171
[1786] https://github.com/STEllAR-GROUP/hpx/pull/3170
[1787] https://github.com/STEllAR-GROUP/hpx/pull/3169
[1788] https://github.com/STEllAR-GROUP/hpx/pull/3168
[1789] https://github.com/STEllAR-GROUP/hpx/issues/3167
[1790] https://github.com/STEllAR-GROUP/hpx/issues/3166
[1791] https://github.com/STEllAR-GROUP/hpx/issues/3165
[1792] https://github.com/STEllAR-GROUP/hpx/pull/3164
[1793] https://github.com/STEllAR-GROUP/hpx/pull/3162
[1794] https://github.com/STEllAR-GROUP/hpx/issues/3161
[1795] https://github.com/STEllAR-GROUP/hpx/pull/3160
[1796] https://github.com/STEllAR-GROUP/hpx/pull/3158
[1797] https://github.com/STEllAR-GROUP/hpx/pull/3157
[1798] https://github.com/STEllAR-GROUP/hpx/pull/3156

- PR #3154[1799] - More refactorings based on clang-tidy reports
- PR #3153[1800] - Fixing a race with timed suspension
- PR #3152[1801] - Documentation for runtime suspension
- PR #3151[1802] - Use small_vector only from boost version 1.59 onwards
- PR #3150[1803] - Avoiding more stack overflows
- PR #3148[1804] - Refactoring component_base and base_action/transfer_base_action
- PR #3147[1805] - Move yield_while out of detail namespace and into own file
- PR #3145[1806] - Remove a leftover of the cxx11 std array cleanup
- PR #3144[1807] - Minor changes to how actions are executed
- PR #3143[1808] - Fix stack overhead
- PR #3142[1809] - Fix typo in config.hpp
- PR #3141[1810] - Fixing small_vector compatibility with older boost version
- PR #3140[1811] - is_heap_text fix
- Issue #3139[1812] - Error in is_heap_tests.hpp
- PR #3138[1813] - Partially reverting #3126
- PR #3137[1814] - Suspend speedup
- PR #3136[1815] - Revert "Fixing #2325"
- PR #3135[1816] - Improving destruction of threads
- Issue #3134[1817] - HPX_SERIALIZATION_SPLIT_FREE does not stop compiler from looking for serialize() method
- PR #3133[1818] - Make hwloc compulsory
- PR #3132[1819] - Update CXX14 constexpr feature test
- PR #3131[1820] - Fixing #2325
- PR #3130[1821] - Avoid completion handler allocation

[1799] https://github.com/STEllAR-GROUP/hpx/pull/3154
[1800] https://github.com/STEllAR-GROUP/hpx/pull/3153
[1801] https://github.com/STEllAR-GROUP/hpx/pull/3152
[1802] https://github.com/STEllAR-GROUP/hpx/pull/3151
[1803] https://github.com/STEllAR-GROUP/hpx/pull/3150
[1804] https://github.com/STEllAR-GROUP/hpx/pull/3148
[1805] https://github.com/STEllAR-GROUP/hpx/pull/3147
[1806] https://github.com/STEllAR-GROUP/hpx/pull/3145
[1807] https://github.com/STEllAR-GROUP/hpx/pull/3144
[1808] https://github.com/STEllAR-GROUP/hpx/pull/3143
[1809] https://github.com/STEllAR-GROUP/hpx/pull/3142
[1810] https://github.com/STEllAR-GROUP/hpx/pull/3141
[1811] https://github.com/STEllAR-GROUP/hpx/pull/3140
[1812] https://github.com/STEllAR-GROUP/hpx/issues/3139
[1813] https://github.com/STEllAR-GROUP/hpx/pull/3138
[1814] https://github.com/STEllAR-GROUP/hpx/pull/3137
[1815] https://github.com/STEllAR-GROUP/hpx/pull/3136
[1816] https://github.com/STEllAR-GROUP/hpx/pull/3135
[1817] https://github.com/STEllAR-GROUP/hpx/issues/3134
[1818] https://github.com/STEllAR-GROUP/hpx/pull/3133
[1819] https://github.com/STEllAR-GROUP/hpx/pull/3132
[1820] https://github.com/STEllAR-GROUP/hpx/pull/3131
[1821] https://github.com/STEllAR-GROUP/hpx/pull/3130

- PR #3129[1822] - Suspend runtime
- PR #3128[1823] - Make docbook dtd and xsl path names consistent
- PR #3127[1824] - Add hpx::start nullptr overloads
- PR #3126[1825] - Cleaning up coroutine implementation
- PR #3125[1826] - Replacing nullptr with hpx::threads::invalid_thread_id
- Issue #3124[1827] - Add hello_world_component to CI builds
- PR #3123[1828] - Add new constructor.
- PR #3122[1829] - Fixing #3121
- Issue #3121[1830] - HPX_SMT_PAUSE is broken on non-x86 platforms when __GNUC__ is defined
- PR #3120[1831] - Don't use boost::intrusive_ptr for thread_id_type
- PR #3119[1832] - Disable default executor compatibility with V1 executors
- PR #3118[1833] - Adding performance_counter::reinit to allow for dynamically changing counter sets
- PR #3117[1834] - Replace uses of boost/experimental::optional with util::optional
- PR #3116[1835] - Moving background thread APEX timer #2980
- PR #3115[1836] - Fixing race condition in channel test
- PR #3114[1837] - Avoid using util::function for thread function wrappers
- PR #3113[1838] - cmake V3.10.2 has changed the variable names used for MPI
- PR #3112[1839] - Minor fixes to exclusive_scan algorithm
- PR #3111[1840] - Revert "fix detection of cxx11_std_atomic"
- PR #3110[1841] - Suspend thread pool
- PR #3109[1842] - Fixing thread scheduling when yielding a thread id
- PR #3108[1843] - Revert "Suspend thread pool"
- PR #3107[1844] - Remove UB from thread::id relational operators

[1822] https://github.com/STEllAR-GROUP/hpx/pull/3129
[1823] https://github.com/STEllAR-GROUP/hpx/pull/3128
[1824] https://github.com/STEllAR-GROUP/hpx/pull/3127
[1825] https://github.com/STEllAR-GROUP/hpx/pull/3126
[1826] https://github.com/STEllAR-GROUP/hpx/pull/3125
[1827] https://github.com/STEllAR-GROUP/hpx/issues/3124
[1828] https://github.com/STEllAR-GROUP/hpx/pull/3123
[1829] https://github.com/STEllAR-GROUP/hpx/pull/3122
[1830] https://github.com/STEllAR-GROUP/hpx/issues/3121
[1831] https://github.com/STEllAR-GROUP/hpx/pull/3120
[1832] https://github.com/STEllAR-GROUP/hpx/pull/3119
[1833] https://github.com/STEllAR-GROUP/hpx/pull/3118
[1834] https://github.com/STEllAR-GROUP/hpx/pull/3117
[1835] https://github.com/STEllAR-GROUP/hpx/pull/3116
[1836] https://github.com/STEllAR-GROUP/hpx/pull/3115
[1837] https://github.com/STEllAR-GROUP/hpx/pull/3114
[1838] https://github.com/STEllAR-GROUP/hpx/pull/3113
[1839] https://github.com/STEllAR-GROUP/hpx/pull/3112
[1840] https://github.com/STEllAR-GROUP/hpx/pull/3111
[1841] https://github.com/STEllAR-GROUP/hpx/pull/3110
[1842] https://github.com/STEllAR-GROUP/hpx/pull/3109
[1843] https://github.com/STEllAR-GROUP/hpx/pull/3108
[1844] https://github.com/STEllAR-GROUP/hpx/pull/3107

- PR #3106[1845] - Add cmake test for std::decay_t to fix cuda build
- PR #3105[1846] - Fixing refcount for async traversal frame
- PR #3104[1847] - Local execution of direct actions is now actually performed directly
- PR #3103[1848] - Adding support for generic counter_raw_values performance counter type
- Issue #3102[1849] - Introduce generic performance counter type returning an array of values
- PR #3101[1850] - Revert "Adapting stack overhead limit for gcc 4.9"
- PR #3100[1851] - Fix #3068 (condition_variable deadlock)
- PR #3099[1852] - Fixing lock held during suspension in papi counter component
- PR #3098[1853] - Unbreak broadcast_wait_for_2822 test
- PR #3097[1854] - Adapting stack overhead limit for gcc 4.9
- PR #3096[1855] - fix detection of cxx11_std_atomic
- PR #3095[1856] - Add ciso646 header to get _LIBCPP_VERSION for testing inplace merge
- PR #3094[1857] - Relax atomic operations on performance counter values
- PR #3093[1858] - Short-circuit all_of/any_of/none_of instantiations
- PR #3092[1859] - Take advantage of C++14 lambda capture initialization syntax, where possible
- PR #3091[1860] - Remove more references to Boost from logging code
- PR #3090[1861] - Unify use of yield/yield_k
- PR #3089[1862] - Fix a strange thing in parallel::detail::handle_exception. (Fix #2834.)
- Issue #3088[1863] - A strange thing in parallel::sort.
- PR #3087[1864] - Fixing assertion in default_distribution_policy
- PR #3086[1865] - Implement parallel::remove and parallel::remove_if
- PR #3085[1866] - Addressing breaking changes in Boost V1.66
- PR #3084[1867] - Ignore build warnings round 2

[1845] https://github.com/STEllAR-GROUP/hpx/pull/3106
[1846] https://github.com/STEllAR-GROUP/hpx/pull/3105
[1847] https://github.com/STEllAR-GROUP/hpx/pull/3104
[1848] https://github.com/STEllAR-GROUP/hpx/pull/3103
[1849] https://github.com/STEllAR-GROUP/hpx/issues/3102
[1850] https://github.com/STEllAR-GROUP/hpx/pull/3101
[1851] https://github.com/STEllAR-GROUP/hpx/pull/3100
[1852] https://github.com/STEllAR-GROUP/hpx/pull/3099
[1853] https://github.com/STEllAR-GROUP/hpx/pull/3098
[1854] https://github.com/STEllAR-GROUP/hpx/pull/3097
[1855] https://github.com/STEllAR-GROUP/hpx/pull/3096
[1856] https://github.com/STEllAR-GROUP/hpx/pull/3095
[1857] https://github.com/STEllAR-GROUP/hpx/pull/3094
[1858] https://github.com/STEllAR-GROUP/hpx/pull/3093
[1859] https://github.com/STEllAR-GROUP/hpx/pull/3092
[1860] https://github.com/STEllAR-GROUP/hpx/pull/3091
[1861] https://github.com/STEllAR-GROUP/hpx/pull/3090
[1862] https://github.com/STEllAR-GROUP/hpx/pull/3089
[1863] https://github.com/STEllAR-GROUP/hpx/issues/3088
[1864] https://github.com/STEllAR-GROUP/hpx/pull/3087
[1865] https://github.com/STEllAR-GROUP/hpx/pull/3086
[1866] https://github.com/STEllAR-GROUP/hpx/pull/3085
[1867] https://github.com/STEllAR-GROUP/hpx/pull/3084

- PR #3083[1868] - Fix typo HPX_WITH_MM_PREFECTH
- PR #3081[1869] - Pre-decay template arguments early
- PR #3080[1870] - Suspend thread pool
- PR #3079[1871] - Ignore build warnings
- PR #3078[1872] - Don't test inplace_merge with libc++
- PR #3076[1873] - Fixing 3075: Part 1
- PR #3074[1874] - Fix more build warnings
- PR #3073[1875] - Suspend thread cleanup
- PR #3072[1876] - Change existing symbol_namespace::iterate to return all data instead of invoking a callback
- PR #3071[1877] - Fixing pack_traversal_async test
- PR #3070[1878] - Fix dynamic_counters_loaded_1508 test by adding dependency to memory_component
- PR #3069[1879] - Fix scheduling loop exit
- Issue #3068[1880] - hpx::lcos::condition_variable could be suspect to deadlocks
- PR #3067[1881] - #ifdef out random_shuffle deprecated in later c++
- PR #3066[1882] - Make coalescing test depend on coalescing library to ensure it gets built
- PR #3065[1883] - Workaround for minimal_timed_async_executor_test compilation failures, attempts to copy a deferred call (in unevaluated context)
- PR #3064[1884] - Fixing wrong condition in wrapper_heap
- PR #3062[1885] - Fix exception handling for execution::seq
- PR #3061[1886] - Adapt MSVC C++ mode handling to VS15.5
- PR #3060[1887] - Fix compiler problem in MSVC release mode
- PR #3059[1888] - Fixing #2931
- Issue #3058[1889] - minimal_timed_async_executor_test_exe fails to compile on master (d6f505c)
- PR #3057[1890] - Fix stable_merge_2964 compilation problems

1868 https://github.com/STEllAR-GROUP/hpx/pull/3083
1869 https://github.com/STEllAR-GROUP/hpx/pull/3081
1870 https://github.com/STEllAR-GROUP/hpx/pull/3080
1871 https://github.com/STEllAR-GROUP/hpx/pull/3079
1872 https://github.com/STEllAR-GROUP/hpx/pull/3078
1873 https://github.com/STEllAR-GROUP/hpx/pull/3076
1874 https://github.com/STEllAR-GROUP/hpx/pull/3074
1875 https://github.com/STEllAR-GROUP/hpx/pull/3073
1876 https://github.com/STEllAR-GROUP/hpx/pull/3072
1877 https://github.com/STEllAR-GROUP/hpx/pull/3071
1878 https://github.com/STEllAR-GROUP/hpx/pull/3070
1879 https://github.com/STEllAR-GROUP/hpx/pull/3069
1880 https://github.com/STEllAR-GROUP/hpx/issues/3068
1881 https://github.com/STEllAR-GROUP/hpx/pull/3067
1882 https://github.com/STEllAR-GROUP/hpx/pull/3066
1883 https://github.com/STEllAR-GROUP/hpx/pull/3065
1884 https://github.com/STEllAR-GROUP/hpx/pull/3064
1885 https://github.com/STEllAR-GROUP/hpx/pull/3062
1886 https://github.com/STEllAR-GROUP/hpx/pull/3061
1887 https://github.com/STEllAR-GROUP/hpx/pull/3060
1888 https://github.com/STEllAR-GROUP/hpx/pull/3059
1889 https://github.com/STEllAR-GROUP/hpx/issues/3058
1890 https://github.com/STEllAR-GROUP/hpx/pull/3057

- PR #3056[1891] - Fix some build warnings caused by unused variables/unnecessary tests
- PR #3055[1892] - Update documentation for running tests
- Issue #3054[1893] - Assertion failure when using bulk hpx::**new_** in asynchronous mode
- PR #3052[1894] - Do not bind test running to cmake test build rule
- PR #3051[1895] - Fix HPX-Qt interaction in Qt example.
- Issue #3048[1896] - nqueen example fails occasionally
- PR #3047[1897] - Fixing #3044
- PR #3046[1898] - Add OS thread suspension
- PR #3042[1899] - PyCicle - first attempt at a build toold for checking PR's
- PR #3041[1900] - Fix a problem about asynchronous execution of parallel::merge and parallel::partition.
- PR #3040[1901] - Fix a mistake about exception handling in asynchronous execution of scan_partitioner.
- PR #3039[1902] - Consistently use executors to schedule work
- PR #3038[1903] - Fixing local direct function execution and lambda actions perfect forwarding
- PR #3035[1904] - Make parallel unit test names match build target/folder names
- PR #3033[1905] - Fix setting of default build type
- Issue #3032[1906] - Fix partitioner arg copy found in #2982
- Issue #3031[1907] - Errors linking libhpx.so due to missing references (master branch, commit 6679a8882)
- PR #3030[1908] - Revert "implement executor then interface with && forwarding reference"
- PR #3029[1909] - Run CI inspect checks before building
- PR #3028[1910] - Added range version of parallel::move
- Issue #3027[1911] - Implement all scheduling APIs in terms of executors
- PR #3026[1912] - implement executor then interface with && forwarding reference
- PR #3025[1913] - Fix typo unitialized to uninitialized

1891 https://github.com/STEllAR-GROUP/hpx/pull/3056
1892 https://github.com/STEllAR-GROUP/hpx/pull/3055
1893 https://github.com/STEllAR-GROUP/hpx/issues/3054
1894 https://github.com/STEllAR-GROUP/hpx/pull/3052
1895 https://github.com/STEllAR-GROUP/hpx/pull/3051
1896 https://github.com/STEllAR-GROUP/hpx/issues/3048
1897 https://github.com/STEllAR-GROUP/hpx/pull/3047
1898 https://github.com/STEllAR-GROUP/hpx/pull/3046
1899 https://github.com/STEllAR-GROUP/hpx/pull/3042
1900 https://github.com/STEllAR-GROUP/hpx/pull/3041
1901 https://github.com/STEllAR-GROUP/hpx/pull/3040
1902 https://github.com/STEllAR-GROUP/hpx/pull/3039
1903 https://github.com/STEllAR-GROUP/hpx/pull/3038
1904 https://github.com/STEllAR-GROUP/hpx/pull/3035
1905 https://github.com/STEllAR-GROUP/hpx/pull/3033
1906 https://github.com/STEllAR-GROUP/hpx/issues/3032
1907 https://github.com/STEllAR-GROUP/hpx/issues/3031
1908 https://github.com/STEllAR-GROUP/hpx/pull/3030
1909 https://github.com/STEllAR-GROUP/hpx/pull/3029
1910 https://github.com/STEllAR-GROUP/hpx/pull/3028
1911 https://github.com/STEllAR-GROUP/hpx/issues/3027
1912 https://github.com/STEllAR-GROUP/hpx/pull/3026
1913 https://github.com/STEllAR-GROUP/hpx/pull/3025

- PR #3024[1914] - Inspect fixes
- PR #3023[1915] - P0356 Simplified partial function application
- PR #3022[1916] - Master fixes
- PR #3021[1917] - Segfault fix
- PR #3020[1918] - Disable command-line aliasing for applications that use user_main
- PR #3019[1919] - Adding enable_elasticity option to pool configuration
- PR #3018[1920] - Fix stack overflow detection configuration in header files
- PR #3017[1921] - Speed up local action execution
- PR #3016[1922] - Unify stack-overflow detection options, remove reference to libsigsegv
- PR #3015[1923] - Speeding up accessing the resource partitioner and the topology info
- Issue #3014[1924] - HPX does not compile on POWER8 with gcc 5.4
- Issue #3013[1925] - hello_world occasionally prints multiple lines from a single OS-thread
- PR #3012[1926] - Silence warning about casting away qualifiers in itt_notify.hpp
- PR #3011[1927] - Fix cpuset leak in hwloc_topology_info.cpp
- PR #3010[1928] - Remove useless decay_copy
- PR #3009[1929] - Fixing 2996
- PR #3008[1930] - Remove unused internal function
- PR #3007[1931] - Fixing wrapper_heap alignment problems
- Issue #3006[1932] - hwloc memory leak
- PR #3004[1933] - Silence C4251 (needs to have dll-interface) for future_data_void
- Issue #3003[1934] - Suspension of runtime
- PR #3001[1935] - Attempting to avoid data races in async_traversal while evaluating dataflow()
- PR #3000[1936] - Adding hpx::util::optional as a first step to replace experimental::optional

[1914] https://github.com/STEllAR-GROUP/hpx/pull/3024
[1915] https://github.com/STEllAR-GROUP/hpx/pull/3023
[1916] https://github.com/STEllAR-GROUP/hpx/pull/3022
[1917] https://github.com/STEllAR-GROUP/hpx/pull/3021
[1918] https://github.com/STEllAR-GROUP/hpx/pull/3020
[1919] https://github.com/STEllAR-GROUP/hpx/pull/3019
[1920] https://github.com/STEllAR-GROUP/hpx/pull/3018
[1921] https://github.com/STEllAR-GROUP/hpx/pull/3017
[1922] https://github.com/STEllAR-GROUP/hpx/pull/3016
[1923] https://github.com/STEllAR-GROUP/hpx/pull/3015
[1924] https://github.com/STEllAR-GROUP/hpx/issues/3014
[1925] https://github.com/STEllAR-GROUP/hpx/issues/3013
[1926] https://github.com/STEllAR-GROUP/hpx/pull/3012
[1927] https://github.com/STEllAR-GROUP/hpx/pull/3011
[1928] https://github.com/STEllAR-GROUP/hpx/pull/3010
[1929] https://github.com/STEllAR-GROUP/hpx/pull/3009
[1930] https://github.com/STEllAR-GROUP/hpx/pull/3008
[1931] https://github.com/STEllAR-GROUP/hpx/pull/3007
[1932] https://github.com/STEllAR-GROUP/hpx/issues/3006
[1933] https://github.com/STEllAR-GROUP/hpx/pull/3004
[1934] https://github.com/STEllAR-GROUP/hpx/issues/3003
[1935] https://github.com/STEllAR-GROUP/hpx/pull/3001
[1936] https://github.com/STEllAR-GROUP/hpx/pull/3000

- PR #2998[1937] - Cleanup up and Fixing component creation and deletion
- Issue #2996[1938] - Build fails with HPX_WITH_HWLOC=OFF
- PR #2995[1939] - Push more future_data functionality to source file
- PR #2994[1940] - WIP: Fix throttle test
- PR #2993[1941] - Making sure –hpx:help does not throw for required (but missing) arguments
- PR #2992[1942] - Adding non-blocking (on destruction) service executors
- Issue #2991[1943] - run_as_os_thread locks up
- Issue #2990[1944] - –help will not work until all required options are provided
- PR #2989[1945] - Improve error messages caused by misuse of dataflow
- PR #2988[1946] - Improve error messages caused by misuse of .then
- Issue #2987[1947] - stack overflow detection producing false positives
- PR #2986[1948] - Deduplicate non-dependent thread_info logging types
- PR #2985[1949] - Adapted parallel::{all_oflany_oflnone_of} for Ranges TS (see #1668)
- PR #2984[1950] - Refactor one_size_heap code to simplify code
- PR #2983[1951] - Fixing local_new_component
- PR #2982[1952] - Clang tidy
- PR #2981[1953] - Simplify allocator rebinding in pack traversal
- PR #2979[1954] - Fixing integer overflows
- PR #2978[1955] - Implement parallel::inplace_merge
- Issue #2977[1956] - Make hwloc compulsory instead of optional
- PR #2976[1957] - Making sure client_base instance that registered the component does not unregister it when being destructed
- PR #2975[1958] - Change version of pulled APEX to master
- PR #2974[1959] - Fix domain not being freed at the end of scheduling loop

1937 https://github.com/STEllAR-GROUP/hpx/pull/2998
1938 https://github.com/STEllAR-GROUP/hpx/issues/2996
1939 https://github.com/STEllAR-GROUP/hpx/pull/2995
1940 https://github.com/STEllAR-GROUP/hpx/pull/2994
1941 https://github.com/STEllAR-GROUP/hpx/pull/2993
1942 https://github.com/STEllAR-GROUP/hpx/pull/2992
1943 https://github.com/STEllAR-GROUP/hpx/issues/2991
1944 https://github.com/STEllAR-GROUP/hpx/issues/2990
1945 https://github.com/STEllAR-GROUP/hpx/pull/2989
1946 https://github.com/STEllAR-GROUP/hpx/pull/2988
1947 https://github.com/STEllAR-GROUP/hpx/issues/2987
1948 https://github.com/STEllAR-GROUP/hpx/pull/2986
1949 https://github.com/STEllAR-GROUP/hpx/pull/2985
1950 https://github.com/STEllAR-GROUP/hpx/pull/2984
1951 https://github.com/STEllAR-GROUP/hpx/pull/2983
1952 https://github.com/STEllAR-GROUP/hpx/pull/2982
1953 https://github.com/STEllAR-GROUP/hpx/pull/2981
1954 https://github.com/STEllAR-GROUP/hpx/pull/2979
1955 https://github.com/STEllAR-GROUP/hpx/pull/2978
1956 https://github.com/STEllAR-GROUP/hpx/issues/2977
1957 https://github.com/STEllAR-GROUP/hpx/pull/2976
1958 https://github.com/STEllAR-GROUP/hpx/pull/2975
1959 https://github.com/STEllAR-GROUP/hpx/pull/2974

- PR #2973[1960] - Fix small typos
- PR #2972[1961] - Adding uintstd.h header
- PR #2971[1962] - Fall back to creating local components using local_new
- PR #2970[1963] - Improve is_tuple_like trait
- PR #2969[1964] - Fix HPX_WITH_MORE_THAN_64_THREADS default value
- PR #2968[1965] - Cleaning up dataflow overload set
- PR #2967[1966] - Make parallel::merge is stable. (Fix #2964.)
- PR #2966[1967] - Fixing a couple of held locks during exception handling
- PR #2965[1968] - Adding missing #include
- Issue #2964[1969] - parallel merge is not stable
- PR #2963[1970] - Making sure any function object passed to dataflow is released after being invoked
- PR #2962[1971] - Partially reverting #2891
- PR #2961[1972] - Attempt to fix the gcc 4.9 problem with the async pack traversal
- Issue #2959[1973] - Program terminates during error handling
- Issue #2958[1974] - HPX_PLAIN_ACTION breaks due to missing include
- PR #2957[1975] - Fixing errors generated by mixing different attribute syntaxes
- Issue #2956[1976] - Mixing attribute syntaxes leads to compiler errors
- Issue #2955[1977] - Fix OS-Thread throttling
- PR #2953[1978] - Making sure any hpx.os_threads=N supplied through a –hpx::config file is taken into account
- PR #2952[1979] - Removing wrong call to cleanup_terminated_locked
- PR #2951[1980] - Revert "Make sure the function vtables are initialized before use"
- PR #2950[1981] - Fix a namespace compilation error when some schedulers are disabled
- Issue #2949[1982] - master branch giving lockups on shutdown

[1960] https://github.com/STEllAR-GROUP/hpx/pull/2973
[1961] https://github.com/STEllAR-GROUP/hpx/pull/2972
[1962] https://github.com/STEllAR-GROUP/hpx/pull/2971
[1963] https://github.com/STEllAR-GROUP/hpx/pull/2970
[1964] https://github.com/STEllAR-GROUP/hpx/pull/2969
[1965] https://github.com/STEllAR-GROUP/hpx/pull/2968
[1966] https://github.com/STEllAR-GROUP/hpx/pull/2967
[1967] https://github.com/STEllAR-GROUP/hpx/pull/2966
[1968] https://github.com/STEllAR-GROUP/hpx/pull/2965
[1969] https://github.com/STEllAR-GROUP/hpx/issues/2964
[1970] https://github.com/STEllAR-GROUP/hpx/pull/2963
[1971] https://github.com/STEllAR-GROUP/hpx/pull/2962
[1972] https://github.com/STEllAR-GROUP/hpx/pull/2961
[1973] https://github.com/STEllAR-GROUP/hpx/issues/2959
[1974] https://github.com/STEllAR-GROUP/hpx/issues/2958
[1975] https://github.com/STEllAR-GROUP/hpx/pull/2957
[1976] https://github.com/STEllAR-GROUP/hpx/issues/2956
[1977] https://github.com/STEllAR-GROUP/hpx/issues/2955
[1978] https://github.com/STEllAR-GROUP/hpx/pull/2953
[1979] https://github.com/STEllAR-GROUP/hpx/pull/2952
[1980] https://github.com/STEllAR-GROUP/hpx/pull/2951
[1981] https://github.com/STEllAR-GROUP/hpx/pull/2950
[1982] https://github.com/STEllAR-GROUP/hpx/issues/2949

- Issue #2947[1983] - hpx.ini is not used correctly at initialization
- PR #2946[1984] - Adding explicit feature test for thread_local
- PR #2945[1985] - Make sure the function vtables are initialized before use
- PR #2944[1986] - Attempting to solve affinity problems on CircleCI
- PR #2943[1987] - Changing channel actions to be direct
- PR #2942[1988] - Adding split_future for std::vector
- PR #2941[1989] - Add a feature test to test for CXX11 override
- Issue #2940[1990] - Add split_future for future<vector<T>>
- PR #2939[1991] - Making error reporting during problems with setting affinity masks more verbose
- PR #2938[1992] - Fix this various executors
- PR #2937[1993] - Fix some typos in documentation
- PR #2934[1994] - Remove the need for "complete" SFINAE checks
- PR #2933[1995] - Making sure parallel::for_loop is executed in parallel if requested
- PR #2932[1996] - Classify chunk_size_iterator to input iterator tag. (Fix #2866)
- Issue #2931[1997] - –hpx:help triggers unusual error with clang build
- PR #2930[1998] - Add #include files needed to set _POSIX_VERSION for debug check
- PR #2929[1999] - Fix a couple of deprecated c++ features
- PR #2928[2000] - Fixing execution parameters
- Issue #2927[2001] - CMake warning: … cycle in constraint graph
- PR #2926[2002] - Default pool rename
- Issue #2925[2003] - Default pool cannot be renamed
- Issue #2924[2004] - hpx:attach-debugger=startup does not work any more
- PR #2923[2005] - Alloc membind

[1983] https://github.com/STEllAR-GROUP/hpx/issues/2947
[1984] https://github.com/STEllAR-GROUP/hpx/pull/2946
[1985] https://github.com/STEllAR-GROUP/hpx/pull/2945
[1986] https://github.com/STEllAR-GROUP/hpx/pull/2944
[1987] https://github.com/STEllAR-GROUP/hpx/pull/2943
[1988] https://github.com/STEllAR-GROUP/hpx/pull/2942
[1989] https://github.com/STEllAR-GROUP/hpx/pull/2941
[1990] https://github.com/STEllAR-GROUP/hpx/issues/2940
[1991] https://github.com/STEllAR-GROUP/hpx/pull/2939
[1992] https://github.com/STEllAR-GROUP/hpx/pull/2938
[1993] https://github.com/STEllAR-GROUP/hpx/pull/2937
[1994] https://github.com/STEllAR-GROUP/hpx/pull/2934
[1995] https://github.com/STEllAR-GROUP/hpx/pull/2933
[1996] https://github.com/STEllAR-GROUP/hpx/pull/2932
[1997] https://github.com/STEllAR-GROUP/hpx/issues/2931
[1998] https://github.com/STEllAR-GROUP/hpx/pull/2930
[1999] https://github.com/STEllAR-GROUP/hpx/pull/2929
[2000] https://github.com/STEllAR-GROUP/hpx/pull/2928
[2001] https://github.com/STEllAR-GROUP/hpx/issues/2927
[2002] https://github.com/STEllAR-GROUP/hpx/pull/2926
[2003] https://github.com/STEllAR-GROUP/hpx/issues/2925
[2004] https://github.com/STEllAR-GROUP/hpx/issues/2924
[2005] https://github.com/STEllAR-GROUP/hpx/pull/2923

- PR #2922[2006] - This fixes CircleCI errors when running with –hpx:bind=none
- PR #2921[2007] - Custom pool executor was missing priority and stacksize options
- PR #2920[2008] - Adding test to trigger problem reported in #2916
- PR #2919[2009] - Make sure the resource_partitioner is properly destructed on hpx::finalize
- Issue #2918[2010] - hpx::init calls wrong (first) callback when called multiple times
- PR #2917[2011] - Adding util::checkpoint
- Issue #2916[2012] - Weird runtime failures when using a channel and chained continuations
- PR #2915[2013] - Introduce executor parameters customization points
- Issue #2914[2014] - Task assignment to current Pool has unintended consequences
- PR #2913[2015] - Fix rp hang
- PR #2912[2016] - Update contributors
- PR #2911[2017] - Fixing CUDA problems
- PR #2910[2018] - Improve error reporting for process component on POSIX systems
- PR #2909[2019] - Fix typo in include path
- PR #2908[2020] - Use proper container according to iterator tag in benchmarks of parallel algorithms
- PR #2907[2021] - Optionally force-delete remaining channel items on close
- PR #2906[2022] - Making sure generated performance counter names are correct
- Issue #2905[2023] - collecting idle-rate performance counters on multiple localities produces an error
- Issue #2904[2024] - build broken for Intel 17 compilers
- PR #2903[2025] - Documentation Updates– Adding New People
- PR #2902[2026] - Fixing service_executor
- PR #2901[2027] - Fixing partitioned_vector creation
- PR #2900[2028] - Add numa-balanced mode to hpx::bind, spread cores over numa domains

2006 https://github.com/STEllAR-GROUP/hpx/pull/2922
2007 https://github.com/STEllAR-GROUP/hpx/pull/2921
2008 https://github.com/STEllAR-GROUP/hpx/pull/2920
2009 https://github.com/STEllAR-GROUP/hpx/pull/2919
2010 https://github.com/STEllAR-GROUP/hpx/issues/2918
2011 https://github.com/STEllAR-GROUP/hpx/pull/2917
2012 https://github.com/STEllAR-GROUP/hpx/issues/2916
2013 https://github.com/STEllAR-GROUP/hpx/pull/2915
2014 https://github.com/STEllAR-GROUP/hpx/issues/2914
2015 https://github.com/STEllAR-GROUP/hpx/pull/2913
2016 https://github.com/STEllAR-GROUP/hpx/pull/2912
2017 https://github.com/STEllAR-GROUP/hpx/pull/2911
2018 https://github.com/STEllAR-GROUP/hpx/pull/2910
2019 https://github.com/STEllAR-GROUP/hpx/pull/2909
2020 https://github.com/STEllAR-GROUP/hpx/pull/2908
2021 https://github.com/STEllAR-GROUP/hpx/pull/2907
2022 https://github.com/STEllAR-GROUP/hpx/pull/2906
2023 https://github.com/STEllAR-GROUP/hpx/issues/2905
2024 https://github.com/STEllAR-GROUP/hpx/issues/2904
2025 https://github.com/STEllAR-GROUP/hpx/pull/2903
2026 https://github.com/STEllAR-GROUP/hpx/pull/2902
2027 https://github.com/STEllAR-GROUP/hpx/pull/2901
2028 https://github.com/STEllAR-GROUP/hpx/pull/2900

- Issue #2899[2029] - hpx::bind does not have a mode that balances cores over numa domains
- PR #2898[2030] - Adding missing #include and missing guard for optional code section
- PR #2897[2031] - Removing dependency on Boost.ICL
- Issue #2896[2032] - Debug build fails without -fpermissive with GCC 7.1 and Boost 1.65
- PR #2895[2033] - Fixing SLURM environment parsing
- PR #2894[2034] - Fix incorrect handling of compile definition with value 0
- Issue #2893[2035] - Disabling schedulers causes build errors
- PR #2892[2036] - added list serializer
- PR #2891[2037] - Resource Partitioner Fixes
- Issue #2890[2038] - Destroying a non-empty channel causes an assertion failure
- PR #2889[2039] - Add check for libatomic
- PR #2888[2040] - Fix compilation problems if HPX_WITH_ITT_NOTIFY=ON
- PR #2887[2041] - Adapt broadcast() to non-unwrapping async<Action>
- PR #2886[2042] - Replace Boost.Random with C++11 <random>
- Issue #2885[2043] - regression in broadcast?
- Issue #2884[2044] - linking `-latomic` is not portable
- PR #2883[2045] - Explicitly set -pthread flag if available
- PR #2882[2046] - Wrap boost::format uses
- Issue #2881[2047] - hpx not compiling with `HPX_WITH_ITTNOTIFY=On`
- Issue #2880[2048] - hpx::bind scatter/balanced give wrong pu masks
- PR #2878[2049] - Fix incorrect pool usage masks setup in RP/thread manager
- PR #2877[2050] - Require `std::array` by default
- PR #2875[2051] - Deprecate use of BOOST_ASSERT

2029 https://github.com/STEllAR-GROUP/hpx/issues/2899
2030 https://github.com/STEllAR-GROUP/hpx/pull/2898
2031 https://github.com/STEllAR-GROUP/hpx/pull/2897
2032 https://github.com/STEllAR-GROUP/hpx/issues/2896
2033 https://github.com/STEllAR-GROUP/hpx/pull/2895
2034 https://github.com/STEllAR-GROUP/hpx/pull/2894
2035 https://github.com/STEllAR-GROUP/hpx/issues/2893
2036 https://github.com/STEllAR-GROUP/hpx/pull/2892
2037 https://github.com/STEllAR-GROUP/hpx/pull/2891
2038 https://github.com/STEllAR-GROUP/hpx/issues/2890
2039 https://github.com/STEllAR-GROUP/hpx/pull/2889
2040 https://github.com/STEllAR-GROUP/hpx/pull/2888
2041 https://github.com/STEllAR-GROUP/hpx/pull/2887
2042 https://github.com/STEllAR-GROUP/hpx/pull/2886
2043 https://github.com/STEllAR-GROUP/hpx/issues/2885
2044 https://github.com/STEllAR-GROUP/hpx/issues/2884
2045 https://github.com/STEllAR-GROUP/hpx/pull/2883
2046 https://github.com/STEllAR-GROUP/hpx/pull/2882
2047 https://github.com/STEllAR-GROUP/hpx/issues/2881
2048 https://github.com/STEllAR-GROUP/hpx/issues/2880
2049 https://github.com/STEllAR-GROUP/hpx/pull/2878
2050 https://github.com/STEllAR-GROUP/hpx/pull/2877
2051 https://github.com/STEllAR-GROUP/hpx/pull/2875

- PR #2874[2052] - Changed serialization of boost.variant to use variadic templates
- Issue #2873[2053] - building with parcelport_mpi fails on cori
- PR #2871[2054] - Adding missing support for throttling scheduler
- PR #2870[2055] - Disambiguate use of base_lco_with_value macros with channel
- Issue #2869[2056] - Difficulty compiling `HPX_REGISTER_CHANNEL_DECLARATION(double)`
- PR #2868[2057] - Removing unneeded assert
- PR #2867[2058] - Implement parallel::unique
- Issue #2866[2059] - The chunk_size_iterator violates multipass guarantee
- PR #2865[2060] - Only use sched_getcpu on linux machines
- PR #2864[2061] - Create redistribution archive for successful builds
- PR #2863[2062] - Replace casts/assignments with hard-coded memcpy operations
- Issue #2862[2063] - sched_getcpu not available on MacOS
- PR #2861[2064] - Fixing unmatched header defines and recursive inclusion of threadmanager
- Issue #2860[2065] - Master program fails with assertion 'type == data_type_address' failed: HPX(assertion_failure)
- Issue #2852[2066] - Support for ARM64
- PR #2858[2067] - Fix misplaced #if #endif's that cause build failure without THREAD_CUMULATIVE_COUNTS
- PR #2857[2068] - Fix some listing in documentation
- PR #2856[2069] - Fixing component handling for lcos
- PR #2855[2070] - Add documentation for coarrays
- PR #2854[2071] - Support ARM64 in timestamps
- PR #2853[2072] - Update Table 17. Non-modifying Parallel Algorithms in Documentation
- PR #2851[2073] - Allowing for non-default-constructible component types

[2052] https://github.com/STEllAR-GROUP/hpx/pull/2874
[2053] https://github.com/STEllAR-GROUP/hpx/issues/2873
[2054] https://github.com/STEllAR-GROUP/hpx/pull/2871
[2055] https://github.com/STEllAR-GROUP/hpx/pull/2870
[2056] https://github.com/STEllAR-GROUP/hpx/issues/2869
[2057] https://github.com/STEllAR-GROUP/hpx/pull/2868
[2058] https://github.com/STEllAR-GROUP/hpx/pull/2867
[2059] https://github.com/STEllAR-GROUP/hpx/issues/2866
[2060] https://github.com/STEllAR-GROUP/hpx/pull/2865
[2061] https://github.com/STEllAR-GROUP/hpx/pull/2864
[2062] https://github.com/STEllAR-GROUP/hpx/pull/2863
[2063] https://github.com/STEllAR-GROUP/hpx/issues/2862
[2064] https://github.com/STEllAR-GROUP/hpx/pull/2861
[2065] https://github.com/STEllAR-GROUP/hpx/issues/2860
[2066] https://github.com/STEllAR-GROUP/hpx/issues/2852
[2067] https://github.com/STEllAR-GROUP/hpx/pull/2858
[2068] https://github.com/STEllAR-GROUP/hpx/pull/2857
[2069] https://github.com/STEllAR-GROUP/hpx/pull/2856
[2070] https://github.com/STEllAR-GROUP/hpx/pull/2855
[2071] https://github.com/STEllAR-GROUP/hpx/pull/2854
[2072] https://github.com/STEllAR-GROUP/hpx/pull/2853
[2073] https://github.com/STEllAR-GROUP/hpx/pull/2851

- PR #2850[2074] - Enable returning future<R> from actions where R is not default-constructible
- PR #2849[2075] - Unify serialization of non-default-constructable types
- Issue #2848[2076] - Components have to be default constructible
- Issue #2847[2077] - Returning a future<R> where R is not default-constructable broken
- Issue #2846[2078] - Unify serialization of non-default-constructible types
- PR #2845[2079] - Add Visual Studio 2015 to the tested toolchains in Appveyor
- Issue #2844[2080] - Change the appveyor build to use the minimal required MSVC version
- Issue #2843[2081] - multi node hello_world hangs
- PR #2842[2082] - Correcting Spelling mistake in docs
- PR #2841[2083] - Fix usage of std::aligned_storage
- PR #2840[2084] - Remove constexpr from a void function
- Issue #2839[2085] - memcpy buffer overflow: load_construct_data() and std::complex members
- Issue #2835[2086] - `constexpr` functions with `void` return type break compilation with CUDA 8.0
- Issue #2834[2087] - One suspicion in parallel::detail::handle_exception
- PR #2833[2088] - Implement parallel::merge
- PR #2832[2089] - Fix a strange thing in parallel::util::detail::handle_local_exceptions. (Fix #2818)
- PR #2830[2090] - Break the debugger when a test failed
- Issue #2831[2091] - `parallel/executors/execution_fwd.hpp` causes compilation failure in C++11 mode.
- PR #2829[2092] - Implement an API for asynchronous pack traversal
- PR #2828[2093] - Split unit test builds on CircleCI to avoid timeouts
- Issue #2827[2094] - failure to compile hello_world example with -Werror
- PR #2824[2095] - Making sure promises are marked as started when used as continuations
- PR #2823[2096] - Add documentation for partitioned_vector_view

[2074] https://github.com/STEllAR-GROUP/hpx/pull/2850
[2075] https://github.com/STEllAR-GROUP/hpx/pull/2849
[2076] https://github.com/STEllAR-GROUP/hpx/issues/2848
[2077] https://github.com/STEllAR-GROUP/hpx/issues/2847
[2078] https://github.com/STEllAR-GROUP/hpx/issues/2846
[2079] https://github.com/STEllAR-GROUP/hpx/pull/2845
[2080] https://github.com/STEllAR-GROUP/hpx/issues/2844
[2081] https://github.com/STEllAR-GROUP/hpx/issues/2843
[2082] https://github.com/STEllAR-GROUP/hpx/pull/2842
[2083] https://github.com/STEllAR-GROUP/hpx/pull/2841
[2084] https://github.com/STEllAR-GROUP/hpx/pull/2840
[2085] https://github.com/STEllAR-GROUP/hpx/issues/2839
[2086] https://github.com/STEllAR-GROUP/hpx/issues/2835
[2087] https://github.com/STEllAR-GROUP/hpx/issues/2834
[2088] https://github.com/STEllAR-GROUP/hpx/pull/2833
[2089] https://github.com/STEllAR-GROUP/hpx/pull/2832
[2090] https://github.com/STEllAR-GROUP/hpx/pull/2830
[2091] https://github.com/STEllAR-GROUP/hpx/issues/2831
[2092] https://github.com/STEllAR-GROUP/hpx/pull/2829
[2093] https://github.com/STEllAR-GROUP/hpx/pull/2828
[2094] https://github.com/STEllAR-GROUP/hpx/issues/2827
[2095] https://github.com/STEllAR-GROUP/hpx/pull/2824
[2096] https://github.com/STEllAR-GROUP/hpx/pull/2823

- Issue #2822[2097] - Yet another issue with wait_for similar to #2796
- PR #2821[2098] - Fix bugs and improve that about HPX_HAVE_CXX11_AUTO_RETURN_VALUE of CMake
- PR #2820[2099] - Support C++11 in benchmark codes of parallel::partition and parallel::partition_copy
- PR #2819[2100] - Fix compile errors in unit test of container version of parallel::partition
- Issue #2818[2101] - A strange thing in parallel::util::detail::handle_local_exceptions
- Issue #2815[2102] - HPX fails to compile with HPX_WITH_CUDA=ON and the new CUDA 9.0 RC
- Issue #2814[2103] - Using 'gmakeN' after 'cmake' produces error in src/CMakeFiles/hpx.dir/runtime/agas/addressing_service.cpp.o
- PR #2813[2104] - Properly support [[noreturn]] attribute if available
- Issue #2812[2105] - Compilation fails with gcc 7.1.1
- PR #2811[2106] - Adding hpx::launch::lazy and support for async, dataflow, and future::then
- PR #2810[2107] - Add option allowing to disable deprecation warning
- PR #2809[2108] - Disable throttling scheduler if HWLOC is not found/used
- PR #2808[2109] - Fix compile errors on some environments of parallel::partition
- Issue #2807[2110] - Difficulty building with `HPX_WITH_HWLOC=Off`
- PR #2806[2111] - Partitioned vector
- PR #2805[2112] - Serializing collections with non-default constructible data
- PR #2802[2113] - Fix FreeBSD 11
- Issue #2801[2114] - Rate limiting techniques in io_service
- Issue #2800[2115] - New Launch Policy: async_if
- PR #2799[2116] - Fix a unit test failure on GCC in tuple_cat
- PR #2798[2117] - bump minimum required cmake to 3.0 in test
- PR #2797[2118] - Making sure future::wait_for et.al. work properly for action results
- Issue #2796[2119] - wait_for does always in "deferred" state for calls on remote localities

2097 https://github.com/STEllAR-GROUP/hpx/issues/2822
2098 https://github.com/STEllAR-GROUP/hpx/pull/2821
2099 https://github.com/STEllAR-GROUP/hpx/pull/2820
2100 https://github.com/STEllAR-GROUP/hpx/pull/2819
2101 https://github.com/STEllAR-GROUP/hpx/issues/2818
2102 https://github.com/STEllAR-GROUP/hpx/issues/2815
2103 https://github.com/STEllAR-GROUP/hpx/issues/2814
2104 https://github.com/STEllAR-GROUP/hpx/pull/2813
2105 https://github.com/STEllAR-GROUP/hpx/issues/2812
2106 https://github.com/STEllAR-GROUP/hpx/pull/2811
2107 https://github.com/STEllAR-GROUP/hpx/pull/2810
2108 https://github.com/STEllAR-GROUP/hpx/pull/2809
2109 https://github.com/STEllAR-GROUP/hpx/pull/2808
2110 https://github.com/STEllAR-GROUP/hpx/issues/2807
2111 https://github.com/STEllAR-GROUP/hpx/pull/2806
2112 https://github.com/STEllAR-GROUP/hpx/pull/2805
2113 https://github.com/STEllAR-GROUP/hpx/pull/2802
2114 https://github.com/STEllAR-GROUP/hpx/issues/2801
2115 https://github.com/STEllAR-GROUP/hpx/issues/2800
2116 https://github.com/STEllAR-GROUP/hpx/pull/2799
2117 https://github.com/STEllAR-GROUP/hpx/pull/2798
2118 https://github.com/STEllAR-GROUP/hpx/pull/2797
2119 https://github.com/STEllAR-GROUP/hpx/issues/2796

- Issue #2795[2120] - Serialization of types without default constructor
- PR #2794[2121] - Fixing test for partitioned_vector iteration
- PR #2792[2122] - Implemented segmented find and its variations for partitioned vector
- PR #2791[2123] - Circumvent scary warning about placement new
- PR #2790[2124] - Fix OSX build
- PR #2789[2125] - Resource partitioner
- PR #2788[2126] - Adapt parallel::is_heap and parallel::is_heap_until to Ranges TS
- PR #2787[2127] - Unwrap hotfixes
- PR #2786[2128] - Update CMake Minimum Version to 3.3.2 (refs #2565)
- Issue #2785[2129] - Issues with masks and cpuset
- PR #2784[2130] - Error with reduce and transform reduce fixed
- PR #2783[2131] - StackOverflow integration with libsigsegv
- PR #2782[2132] - Replace boost::atomic with std::atomic (where possible)
- PR #2781[2133] - Check for and optionally use [[deprecated]] attribute
- PR #2780[2134] - Adding empty (but non-trivial) destructor to circumvent warnings
- PR #2779[2135] - Exception info tweaks
- PR #2778[2136] - Implement parallel::partition
- PR #2777[2137] - Improve error handling in gather_here/gather_there
- PR #2776[2138] - Fix a bug in compiler version check
- PR #2775[2139] - Fix compilation when HPX_WITH_LOGGING is OFF
- PR #2774[2140] - Removing dependency on Boost.Date_Time
- PR #2773[2141] - Add sync_images() method to spmd_block class
- PR #2772[2142] - Adding documentation for PAPI counters

[2120] https://github.com/STEllAR-GROUP/hpx/issues/2795
[2121] https://github.com/STEllAR-GROUP/hpx/pull/2794
[2122] https://github.com/STEllAR-GROUP/hpx/pull/2792
[2123] https://github.com/STEllAR-GROUP/hpx/pull/2791
[2124] https://github.com/STEllAR-GROUP/hpx/pull/2790
[2125] https://github.com/STEllAR-GROUP/hpx/pull/2789
[2126] https://github.com/STEllAR-GROUP/hpx/pull/2788
[2127] https://github.com/STEllAR-GROUP/hpx/pull/2787
[2128] https://github.com/STEllAR-GROUP/hpx/pull/2786
[2129] https://github.com/STEllAR-GROUP/hpx/issues/2785
[2130] https://github.com/STEllAR-GROUP/hpx/pull/2784
[2131] https://github.com/STEllAR-GROUP/hpx/pull/2783
[2132] https://github.com/STEllAR-GROUP/hpx/pull/2782
[2133] https://github.com/STEllAR-GROUP/hpx/pull/2781
[2134] https://github.com/STEllAR-GROUP/hpx/pull/2780
[2135] https://github.com/STEllAR-GROUP/hpx/pull/2779
[2136] https://github.com/STEllAR-GROUP/hpx/pull/2778
[2137] https://github.com/STEllAR-GROUP/hpx/pull/2777
[2138] https://github.com/STEllAR-GROUP/hpx/pull/2776
[2139] https://github.com/STEllAR-GROUP/hpx/pull/2775
[2140] https://github.com/STEllAR-GROUP/hpx/pull/2774
[2141] https://github.com/STEllAR-GROUP/hpx/pull/2773
[2142] https://github.com/STEllAR-GROUP/hpx/pull/2772

- PR #2771[2143] - Removing boost preprocessor dependency
- PR #2770[2144] - Adding test, fixing deadlock in config registry
- PR #2769[2145] - Remove some other warnings and errors detected by clang 5.0
- Issue #2768[2146] - Is there iterator tag for HPX?
- PR #2767[2147] - Improvements to continuation annotation
- PR #2765[2148] - gcc split stack support for HPX threads #620
- PR #2764[2149] - Fix some uses of begin/end, remove unnecessary includes
- PR #2763[2150] - Bump minimal Boost version to 1.55.0
- PR #2762[2151] - hpx::partitioned_vector serializer
- PR #2761[2152] - Adding configuration summary to cmake output and –hpx:info
- PR #2760[2153] - Removing 1d_hydro example as it is broken
- PR #2758[2154] - Remove various warnings detected by clang 5.0
- Issue #2757[2155] - In case of a "raw thread" is needed per core for implementing parallel algorithm, what is good practice in HPX?
- PR #2756[2156] - Allowing for LCOs to be simple components
- PR #2755[2157] - Removing make_index_pack_unrolled
- PR #2754[2158] - Implement parallel::unique_copy
- PR #2753[2159] - Fixing detection of [[fallthrough]] attribute
- PR #2752[2160] - New thread priority names
- PR #2751[2161] - Replace boost::exception with proposed exception_info
- PR #2750[2162] - Replace boost::iterator_range
- PR #2749[2163] - Fixing hdf5 examples
- Issue #2748[2164] - HPX fails to build with enabled hdf5 examples
- Issue #2747[2165] - Inherited task priorities break certain DAG optimizations

2143 https://github.com/STEllAR-GROUP/hpx/pull/2771
2144 https://github.com/STEllAR-GROUP/hpx/pull/2770
2145 https://github.com/STEllAR-GROUP/hpx/pull/2769
2146 https://github.com/STEllAR-GROUP/hpx/issues/2768
2147 https://github.com/STEllAR-GROUP/hpx/pull/2767
2148 https://github.com/STEllAR-GROUP/hpx/pull/2765
2149 https://github.com/STEllAR-GROUP/hpx/pull/2764
2150 https://github.com/STEllAR-GROUP/hpx/pull/2763
2151 https://github.com/STEllAR-GROUP/hpx/pull/2762
2152 https://github.com/STEllAR-GROUP/hpx/pull/2761
2153 https://github.com/STEllAR-GROUP/hpx/pull/2760
2154 https://github.com/STEllAR-GROUP/hpx/pull/2758
2155 https://github.com/STEllAR-GROUP/hpx/issues/2757
2156 https://github.com/STEllAR-GROUP/hpx/pull/2756
2157 https://github.com/STEllAR-GROUP/hpx/pull/2755
2158 https://github.com/STEllAR-GROUP/hpx/pull/2754
2159 https://github.com/STEllAR-GROUP/hpx/pull/2753
2160 https://github.com/STEllAR-GROUP/hpx/pull/2752
2161 https://github.com/STEllAR-GROUP/hpx/pull/2751
2162 https://github.com/STEllAR-GROUP/hpx/pull/2750
2163 https://github.com/STEllAR-GROUP/hpx/pull/2749
2164 https://github.com/STEllAR-GROUP/hpx/issues/2748
2165 https://github.com/STEllAR-GROUP/hpx/issues/2747

- Issue #2746[2166] - HPX segfaulting with valgrind
- PR #2745[2167] - Adding extended arithmetic performance counters
- PR #2744[2168] - Adding ability to statistics counters to reset base counter
- Issue #2743[2169] - Statistics counter does not support resetting
- PR #2742[2170] - Making sure Vc V2 builds without additional HPX configuration flags
- PR #2741[2171] - Deprecate unwrapped and implement unwrap and unwrapping
- PR #2740[2172] - Coroutine stackoverflow detection for linux/posix; Issue #2408
- PR #2739[2173] - Add files via upload
- PR #2738[2174] - Appveyor support
- PR #2737[2175] - Fixing 2735
- Issue #2736[2176] - 1d_hydro example doesn't work
- Issue #2735[2177] - partitioned_vector_subview test failing
- PR #2734[2178] - Add C++11 range utilities
- PR #2733[2179] - Adapting iterator requirements for parallel algorithms
- PR #2732[2180] - Integrate C++ Co-arrays
- PR #2731[2181] - Adding on_migrated event handler to migratable component instances
- Issue #2729[2182] - Add on_migrated() event handler to migratable components
- Issue #2728[2183] - Why Projection is needed in parallel algorithms?
- PR #2727[2184] - Cmake files for StackOverflow Detection
- PR #2726[2185] - CMake for Stack Overflow Detection
- PR #2725[2186] - Implemented segmented algorithms for partitioned vector
- PR #2724[2187] - Fix examples in Action documentation
- PR #2723[2188] - Enable lcos::channel<T>::register_as

2166 https://github.com/STEllAR-GROUP/hpx/issues/2746
2167 https://github.com/STEllAR-GROUP/hpx/pull/2745
2168 https://github.com/STEllAR-GROUP/hpx/pull/2744
2169 https://github.com/STEllAR-GROUP/hpx/issues/2743
2170 https://github.com/STEllAR-GROUP/hpx/pull/2742
2171 https://github.com/STEllAR-GROUP/hpx/pull/2741
2172 https://github.com/STEllAR-GROUP/hpx/pull/2740
2173 https://github.com/STEllAR-GROUP/hpx/pull/2739
2174 https://github.com/STEllAR-GROUP/hpx/pull/2738
2175 https://github.com/STEllAR-GROUP/hpx/pull/2737
2176 https://github.com/STEllAR-GROUP/hpx/issues/2736
2177 https://github.com/STEllAR-GROUP/hpx/issues/2735
2178 https://github.com/STEllAR-GROUP/hpx/pull/2734
2179 https://github.com/STEllAR-GROUP/hpx/pull/2733
2180 https://github.com/STEllAR-GROUP/hpx/pull/2732
2181 https://github.com/STEllAR-GROUP/hpx/pull/2731
2182 https://github.com/STEllAR-GROUP/hpx/issues/2729
2183 https://github.com/STEllAR-GROUP/hpx/issues/2728
2184 https://github.com/STEllAR-GROUP/hpx/pull/2727
2185 https://github.com/STEllAR-GROUP/hpx/pull/2726
2186 https://github.com/STEllAR-GROUP/hpx/pull/2725
2187 https://github.com/STEllAR-GROUP/hpx/pull/2724
2188 https://github.com/STEllAR-GROUP/hpx/pull/2723

- Issue #2722[2189] - channel register_as() failing on compilation
- PR #2721[2190] - Mind map
- PR #2720[2191] - reorder forward declarations to get rid of C++14-only auto return types
- PR #2719[2192] - Add documentation for partitioned_vector and add features in pack.hpp
- Issue #2718[2193] - Some forward declarations in execution_fwd.hpp aren't C++11-compatible
- PR #2717[2194] - Config support for fallthrough attribute
- PR #2716[2195] - Implement parallel::partition_copy
- PR #2715[2196] - initial import of icu string serializer
- PR #2714[2197] - initial import of valarray serializer
- PR #2713[2198] - Remove slashes before CMAKE_FILES_DIRECTORY variables
- PR #2712[2199] - Fixing wait for 1751
- PR #2711[2200] - Adjust code for minimal supported GCC having being bumped to 4.9
- PR #2710[2201] - Adding code of conduct
- PR #2709[2202] - Fixing UB in destroy tests
- PR #2708[2203] - Add inline to prevent multiple definition issue
- Issue #2707[2204] - Multiple defined symbols for task_block.hpp in VS2015
- PR #2706[2205] - Adding .clang-format file
- PR #2704[2206] - Add a synchronous mapping API
- Issue #2703[2207] - Request: Add the .clang-format file to the repository
- Issue #2702[2208] - STEllAR-GROUP/Vc slower than VCv1 possibly due to wrong instructions generated
- Issue #2701[2209] - Datapar with STEllAR-GROUP/Vc requires obscure flag
- Issue #2700[2210] - Naming inconsistency in parallel algorithms
- Issue #2699[2211] - Iterator requirements are different from standard in parallel copy_if.

2189 https://github.com/STEllAR-GROUP/hpx/issues/2722
2190 https://github.com/STEllAR-GROUP/hpx/pull/2721
2191 https://github.com/STEllAR-GROUP/hpx/pull/2720
2192 https://github.com/STEllAR-GROUP/hpx/pull/2719
2193 https://github.com/STEllAR-GROUP/hpx/issues/2718
2194 https://github.com/STEllAR-GROUP/hpx/pull/2717
2195 https://github.com/STEllAR-GROUP/hpx/pull/2716
2196 https://github.com/STEllAR-GROUP/hpx/pull/2715
2197 https://github.com/STEllAR-GROUP/hpx/pull/2714
2198 https://github.com/STEllAR-GROUP/hpx/pull/2713
2199 https://github.com/STEllAR-GROUP/hpx/pull/2712
2200 https://github.com/STEllAR-GROUP/hpx/pull/2711
2201 https://github.com/STEllAR-GROUP/hpx/pull/2710
2202 https://github.com/STEllAR-GROUP/hpx/pull/2709
2203 https://github.com/STEllAR-GROUP/hpx/pull/2708
2204 https://github.com/STEllAR-GROUP/hpx/issues/2707
2205 https://github.com/STEllAR-GROUP/hpx/pull/2706
2206 https://github.com/STEllAR-GROUP/hpx/pull/2704
2207 https://github.com/STEllAR-GROUP/hpx/issues/2703
2208 https://github.com/STEllAR-GROUP/hpx/issues/2702
2209 https://github.com/STEllAR-GROUP/hpx/issues/2701
2210 https://github.com/STEllAR-GROUP/hpx/issues/2700
2211 https://github.com/STEllAR-GROUP/hpx/issues/2699

- PR #2698[2212] - Properly releasing parcelport write handlers
- Issue #2697[2213] - Compile error in addressing_service.cpp
- Issue #2696[2214] - Building and using HPX statically: undefined references from runtime_support_server.cpp
- Issue #2695[2215] - Executor changes cause compilation failures
- PR #2694[2216] - Refining C++ language mode detection for MSVC
- PR #2693[2217] - P0443 r2
- PR #2692[2218] - Partially reverting changes to parcel_await
- Issue #2689[2219] - HPX build fails when HPX_WITH_CUDA is enabled
- PR #2688[2220] - Make Cuda Clang builds pass
- PR #2687[2221] - Add an is_tuple_like trait for sequenceable type detection
- PR #2686[2222] - Allowing throttling scheduler to be used without idle backoff
- PR #2685[2223] - Add support of std::array to hpx::util::tuple_size and tuple_element
- PR #2684[2224] - Adding new statistics performance counters
- PR #2683[2225] - Replace boost::exception_ptr with std::exception_ptr
- Issue #2682[2226] - HPX does not compile with HPX_WITH_THREAD_MANAGER_IDLE_BACKOFF=OFF
- PR #2681[2227] - Attempt to fix problem in managed_component_base
- PR #2680[2228] - Fix bad size during archive creation
- Issue #2679[2229] - Mismatch between size of archive and container
- Issue #2678[2230] - In parallel algorithm, other tasks are executed to the end even if an exception occurs in any task.
- PR #2677[2231] - Adding include check for std::addressof
- PR #2676[2232] - Adding parallel::destroy and destroy_n
- PR #2675[2233] - Making sure statistics counters work as expected
- PR #2674[2234] - Turning assertions into exceptions

[2212] https://github.com/STEllAR-GROUP/hpx/pull/2698
[2213] https://github.com/STEllAR-GROUP/hpx/issues/2697
[2214] https://github.com/STEllAR-GROUP/hpx/issues/2696
[2215] https://github.com/STEllAR-GROUP/hpx/issues/2695
[2216] https://github.com/STEllAR-GROUP/hpx/pull/2694
[2217] https://github.com/STEllAR-GROUP/hpx/pull/2693
[2218] https://github.com/STEllAR-GROUP/hpx/pull/2692
[2219] https://github.com/STEllAR-GROUP/hpx/issues/2689
[2220] https://github.com/STEllAR-GROUP/hpx/pull/2688
[2221] https://github.com/STEllAR-GROUP/hpx/pull/2687
[2222] https://github.com/STEllAR-GROUP/hpx/pull/2686
[2223] https://github.com/STEllAR-GROUP/hpx/pull/2685
[2224] https://github.com/STEllAR-GROUP/hpx/pull/2684
[2225] https://github.com/STEllAR-GROUP/hpx/pull/2683
[2226] https://github.com/STEllAR-GROUP/hpx/issues/2682
[2227] https://github.com/STEllAR-GROUP/hpx/pull/2681
[2228] https://github.com/STEllAR-GROUP/hpx/pull/2680
[2229] https://github.com/STEllAR-GROUP/hpx/issues/2679
[2230] https://github.com/STEllAR-GROUP/hpx/issues/2678
[2231] https://github.com/STEllAR-GROUP/hpx/pull/2677
[2232] https://github.com/STEllAR-GROUP/hpx/pull/2676
[2233] https://github.com/STEllAR-GROUP/hpx/pull/2675
[2234] https://github.com/STEllAR-GROUP/hpx/pull/2674

- PR #2673[2235] - Inhibit direct conversion from future<future<T>> –> future<void>
- PR #2672[2236] - C++17 invoke forms
- PR #2671[2237] - Adding uninitialized_value_construct and uninitialized_value_construct_n
- PR #2670[2238] - Integrate spmd multidimensional views for partitioned_vectors
- PR #2669[2239] - Adding uninitialized_default_construct and uninitialized_default_construct_n
- PR #2668[2240] - Fixing documentation index
- Issue #2667[2241] - Ambiguity of nested hpx::future<void>'s
- Issue #2666[2242] - Statistics Performance counter is not working
- PR #2664[2243] - Adding uninitialized_move and uninitialized_move_n
- Issue #2663[2244] - Seg fault in managed_component::get_base_gid, possibly cause by util::reinitializable_static
- Issue #2662[2245] - Crash in managed_component::get_base_gid due to problem with util::reinitializable_static
- PR #2665[2246] - Hide the `detail` namespace in doxygen per default
- PR #2660[2247] - Add documentation to hpx::util::unwrapped and hpx::util::unwrapped2
- PR #2659[2248] - Improve integration with vcpkg
- PR #2658[2249] - Unify access_data trait for use in both, serialization and de-serialization
- PR #2657[2250] - Removing hpx::lcos::queue<T>
- PR #2656[2251] - Reduce MAX_TERMINATED_THREADS default, improve memory use on manycore cpus
- PR #2655[2252] - Mainteinance for emulate-deleted macros
- PR #2654[2253] - Implement parallel is_heap and is_heap_until
- PR #2653[2254] - Drop support for VS2013
- PR #2652[2255] - This patch makes sure that all parcels in a batch are properly handled
- PR #2649[2256] - Update docs (Table 18) - move transform to end
- Issue #2647[2257] - hpx::parcelset::detail::parcel_data::**has_continuation_** is uninitialized

[2235] https://github.com/STEllAR-GROUP/hpx/pull/2673
[2236] https://github.com/STEllAR-GROUP/hpx/pull/2672
[2237] https://github.com/STEllAR-GROUP/hpx/pull/2671
[2238] https://github.com/STEllAR-GROUP/hpx/pull/2670
[2239] https://github.com/STEllAR-GROUP/hpx/pull/2669
[2240] https://github.com/STEllAR-GROUP/hpx/pull/2668
[2241] https://github.com/STEllAR-GROUP/hpx/issues/2667
[2242] https://github.com/STEllAR-GROUP/hpx/issues/2666
[2243] https://github.com/STEllAR-GROUP/hpx/pull/2664
[2244] https://github.com/STEllAR-GROUP/hpx/issues/2663
[2245] https://github.com/STEllAR-GROUP/hpx/issues/2662
[2246] https://github.com/STEllAR-GROUP/hpx/pull/2665
[2247] https://github.com/STEllAR-GROUP/hpx/pull/2660
[2248] https://github.com/STEllAR-GROUP/hpx/pull/2659
[2249] https://github.com/STEllAR-GROUP/hpx/pull/2658
[2250] https://github.com/STEllAR-GROUP/hpx/pull/2657
[2251] https://github.com/STEllAR-GROUP/hpx/pull/2656
[2252] https://github.com/STEllAR-GROUP/hpx/pull/2655
[2253] https://github.com/STEllAR-GROUP/hpx/pull/2654
[2254] https://github.com/STEllAR-GROUP/hpx/pull/2653
[2255] https://github.com/STEllAR-GROUP/hpx/pull/2652
[2256] https://github.com/STEllAR-GROUP/hpx/pull/2649
[2257] https://github.com/STEllAR-GROUP/hpx/issues/2647

- Issue #2644[2258] - Some .vcxproj in the HPX.sln fail to build
- Issue #2641[2259] - `hpx::lcos::queue` should be deprecated
- PR #2640[2260] - A new throttling policy with public APIs to suspend/resume
- PR #2639[2261] - Fix a tiny typo in tutorial.
- Issue #2638[2262] - Invalid return type 'void' of constexpr function
- PR #2636[2263] - Add and use HPX_MSVC_WARNING_PRAGMA for #pragma warning
- PR #2633[2264] - Distributed define_spmd_block
- PR #2632[2265] - Making sure container serialization uses size-compatible types
- PR #2631[2266] - Add lcos::local::one_element_channel
- PR #2629[2267] - Move unordered_map out of parcelport into hpx/concurrent
- PR #2628[2268] - Making sure that shutdown does not hang
- PR #2627[2269] - Fix serialization
- PR #2626[2270] - Generate `cmake_variables.qbk` and `cmake_toolchains.qbk` outside of the source tree
- PR #2625[2271] - Supporting -std=c++17 flag
- PR #2624[2272] - Fixing a small cmake typo
- PR #2622[2273] - Update CMake minimum required version to 3.0.2 (closes #2621)
- Issue #2621[2274] - Compiling hpx master fails with /usr/bin/ld: final link failed: Bad value
- PR #2620[2275] - Remove warnings due to some captured variables
- PR #2619[2276] - LF multiple parcels
- PR #2618[2277] - Some fixes to libfabric that didn't get caught before the merge
- PR #2617[2278] - Adding `hpx::local_new`
- PR #2616[2279] - Documentation: Extract all entities in order to autolink functions correctly
- Issue #2615[2280] - Documentation: Linking functions is broken

2258 https://github.com/STEllAR-GROUP/hpx/issues/2644
2259 https://github.com/STEllAR-GROUP/hpx/issues/2641
2260 https://github.com/STEllAR-GROUP/hpx/pull/2640
2261 https://github.com/STEllAR-GROUP/hpx/pull/2639
2262 https://github.com/STEllAR-GROUP/hpx/issues/2638
2263 https://github.com/STEllAR-GROUP/hpx/pull/2636
2264 https://github.com/STEllAR-GROUP/hpx/pull/2633
2265 https://github.com/STEllAR-GROUP/hpx/pull/2632
2266 https://github.com/STEllAR-GROUP/hpx/pull/2631
2267 https://github.com/STEllAR-GROUP/hpx/pull/2629
2268 https://github.com/STEllAR-GROUP/hpx/pull/2628
2269 https://github.com/STEllAR-GROUP/hpx/pull/2627
2270 https://github.com/STEllAR-GROUP/hpx/pull/2626
2271 https://github.com/STEllAR-GROUP/hpx/pull/2625
2272 https://github.com/STEllAR-GROUP/hpx/pull/2624
2273 https://github.com/STEllAR-GROUP/hpx/pull/2622
2274 https://github.com/STEllAR-GROUP/hpx/issues/2621
2275 https://github.com/STEllAR-GROUP/hpx/pull/2620
2276 https://github.com/STEllAR-GROUP/hpx/pull/2619
2277 https://github.com/STEllAR-GROUP/hpx/pull/2618
2278 https://github.com/STEllAR-GROUP/hpx/pull/2617
2279 https://github.com/STEllAR-GROUP/hpx/pull/2616
2280 https://github.com/STEllAR-GROUP/hpx/issues/2615

- PR #2614[2281] - Adding serialization for std::deque
- PR #2613[2282] - We need to link with boost.thread and boost.chrono if we use boost.context
- PR #2612[2283] - Making sure for_loop_n(par, …) is actually executed in parallel
- PR #2611[2284] - Add documentation to invoke_fused and friends NFC
- PR #2610[2285] - Added reduction templates using an identity value
- PR #2608[2286] - Fixing some unused vars in inspect
- PR #2607[2287] - Fixed build for mingw
- PR #2606[2288] - Supporting generic context for boost >= 1.61
- PR #2605[2289] - Parcelport libfabric3
- PR #2604[2290] - Adding allocator support to promise and friends
- PR #2603[2291] - Barrier hang
- PR #2602[2292] - Changes to scheduler to steal from one high-priority queue
- Issue #2601[2293] - High priority tasks are not executed first
- PR #2600[2294] - Compat fixes
- PR #2599[2295] - Compatibility layer for threading support
- PR #2598[2296] - V1.1
- PR #2597[2297] - Release V1.0
- PR #2592[2298] - First attempt to introduce spmd_block in hpx
- PR #2586[2299] - local_segment in segmented_iterator_traits
- Issue #2584[2300] - Add allocator support to promise, packaged_task and friends
- PR #2576[2301] - Add missing dependencies of cuda based tests
- PR #2575[2302] - Remove warnings due to some captured variables
- Issue #2574[2303] - MSVC 2015 Compiler crash when building HPX

[2281] https://github.com/STEllAR-GROUP/hpx/pull/2614
[2282] https://github.com/STEllAR-GROUP/hpx/pull/2613
[2283] https://github.com/STEllAR-GROUP/hpx/pull/2612
[2284] https://github.com/STEllAR-GROUP/hpx/pull/2611
[2285] https://github.com/STEllAR-GROUP/hpx/pull/2610
[2286] https://github.com/STEllAR-GROUP/hpx/pull/2608
[2287] https://github.com/STEllAR-GROUP/hpx/pull/2607
[2288] https://github.com/STEllAR-GROUP/hpx/pull/2606
[2289] https://github.com/STEllAR-GROUP/hpx/pull/2605
[2290] https://github.com/STEllAR-GROUP/hpx/pull/2604
[2291] https://github.com/STEllAR-GROUP/hpx/pull/2603
[2292] https://github.com/STEllAR-GROUP/hpx/pull/2602
[2293] https://github.com/STEllAR-GROUP/hpx/issues/2601
[2294] https://github.com/STEllAR-GROUP/hpx/pull/2600
[2295] https://github.com/STEllAR-GROUP/hpx/pull/2599
[2296] https://github.com/STEllAR-GROUP/hpx/pull/2598
[2297] https://github.com/STEllAR-GROUP/hpx/pull/2597
[2298] https://github.com/STEllAR-GROUP/hpx/pull/2592
[2299] https://github.com/STEllAR-GROUP/hpx/pull/2586
[2300] https://github.com/STEllAR-GROUP/hpx/issues/2584
[2301] https://github.com/STEllAR-GROUP/hpx/pull/2576
[2302] https://github.com/STEllAR-GROUP/hpx/pull/2575
[2303] https://github.com/STEllAR-GROUP/hpx/issues/2574

- Issue #2568[2304] - Remove throttle_scheduler as it has been abandoned
- Issue #2566[2305] - Add an inline versioning namespace before 1.0 release
- Issue #2565[2306] - Raise minimal cmake version requirement
- PR #2556[2307] - Fixing scan partitioner
- PR #2546[2308] - Broadcast async
- Issue #2543[2309] - make install fails due to a non-existing .so file
- PR #2495[2310] - wait_or_add_new returning thread_id_type
- Issue #2480[2311] - Unable to register new performance counter
- Issue #2471[2312] - no type named 'fcontext_t' in namespace
- Issue #2456[2313] - Re-implement hpx::util::unwrapped
- Issue #2455[2314] - Add more arithmetic performance counters
- PR #2454[2315] - Fix a couple of warnings and compiler errors
- PR #2453[2316] - Timed executor support
- PR #2447[2317] - Implementing new executor API (P0443)
- Issue #2439[2318] - Implement executor proposal
- Issue #2408[2319] - Stackoverflow detection for linux, e.g. based on libsigsegv
- PR #2377[2320] - Add a customization point for put_parcel so we can override actions
- Issue #2368[2321] - HPX_ASSERT problem
- Issue #2324[2322] - Change default number of threads used to the maximum of the system
- Issue #2266[2323] - hpx_0.9.99 make tests fail
- PR #2195[2324] - Support for code completion in VIM
- Issue #2137[2325] - Hpx does not compile over osx
- Issue #2092[2326] - make tests should just build the tests

2304 https://github.com/STEllAR-GROUP/hpx/issues/2568
2305 https://github.com/STEllAR-GROUP/hpx/issues/2566
2306 https://github.com/STEllAR-GROUP/hpx/issues/2565
2307 https://github.com/STEllAR-GROUP/hpx/pull/2556
2308 https://github.com/STEllAR-GROUP/hpx/pull/2546
2309 https://github.com/STEllAR-GROUP/hpx/issues/2543
2310 https://github.com/STEllAR-GROUP/hpx/pull/2495
2311 https://github.com/STEllAR-GROUP/hpx/issues/2480
2312 https://github.com/STEllAR-GROUP/hpx/issues/2471
2313 https://github.com/STEllAR-GROUP/hpx/issues/2456
2314 https://github.com/STEllAR-GROUP/hpx/issues/2455
2315 https://github.com/STEllAR-GROUP/hpx/pull/2454
2316 https://github.com/STEllAR-GROUP/hpx/pull/2453
2317 https://github.com/STEllAR-GROUP/hpx/pull/2447
2318 https://github.com/STEllAR-GROUP/hpx/issues/2439
2319 https://github.com/STEllAR-GROUP/hpx/issues/2408
2320 https://github.com/STEllAR-GROUP/hpx/pull/2377
2321 https://github.com/STEllAR-GROUP/hpx/issues/2368
2322 https://github.com/STEllAR-GROUP/hpx/issues/2324
2323 https://github.com/STEllAR-GROUP/hpx/issues/2266
2324 https://github.com/STEllAR-GROUP/hpx/pull/2195
2325 https://github.com/STEllAR-GROUP/hpx/issues/2137
2326 https://github.com/STEllAR-GROUP/hpx/issues/2092

- Issue #2026[2327] - Build HPX with Apple's clang
- Issue #1932[2328] - hpx with PBS fails on multiple localities
- PR #1914[2329] - Parallel heap algorithm implementations WIP
- Issue #1598[2330] - Disconnecting a locality results in segfault using heartbeat example
- Issue #1404[2331] - unwrapped doesn't work with movable only types
- Issue #1400[2332] - hpx::util::unwrapped doesn't work with non-future types
- Issue #1205[2333] - TSS is broken
- Issue #1126[2334] - vector<future<T> > does not work gracefully with dataflow, when_all and unwrapped
- Issue #1056[2335] - Thread manager cleanup
- Issue #863[2336] - Futures should not require a default constructor
- Issue #856[2337] - Allow runtimemode_connect to be used with security enabled
- Issue #726[2338] - Valgrind
- Issue #701[2339] - Add RCR performance counter component
- Issue #528[2340] - Add support for known failures and warning count/comparisons to hpx_run_tests.py

2.11.9 *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[2341] 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.

2327 https://github.com/STEllAR-GROUP/hpx/issues/2026
2328 https://github.com/STEllAR-GROUP/hpx/issues/1932
2329 https://github.com/STEllAR-GROUP/hpx/pull/1914
2330 https://github.com/STEllAR-GROUP/hpx/issues/1598
2331 https://github.com/STEllAR-GROUP/hpx/issues/1404
2332 https://github.com/STEllAR-GROUP/hpx/issues/1400
2333 https://github.com/STEllAR-GROUP/hpx/issues/1205
2334 https://github.com/STEllAR-GROUP/hpx/issues/1126
2335 https://github.com/STEllAR-GROUP/hpx/issues/1056
2336 https://github.com/STEllAR-GROUP/hpx/issues/863
2337 https://github.com/STEllAR-GROUP/hpx/issues/856
2338 https://github.com/STEllAR-GROUP/hpx/issues/726
2339 https://github.com/STEllAR-GROUP/hpx/issues/701
2340 https://github.com/STEllAR-GROUP/hpx/issues/528
2341 https://github.com/STEllAR-GROUP/hpx/pull/2289

- We have started to add support for vectorization to our parallel algorithm implementations. This support depends on using an external library, currently either Vc Library or |boost_simd|_. Please see Issue #2333[2342] 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[2343] for more information.
- The parameter sequence for the `hpx::parallel::transform_reduce` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17. The old overload can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.
- The algorithm `hpx::parallel::inner_product` has been renamed to `hpx::parallel::transform_reduce` to match the changes this algorithm has undergone while being moved to C++17. The old inner_product names can be still enabled at configure time by specifying `-DHPX_WITH_TRANSFORM_REDUCE_COMPATIBILITY=On` to CMake.
- Added versions of `hpx::get_ptr` taking client side representations for component instances as their parameter (instead of a global id).
- Added the helper utility `hpx::performance_counters::performance_counter_set` helping to encapsulate a set of performance counters to be managed concurrently.
- All execution policies and related classes have been renamed to be consistent with the naming changes applied for C++17. All policies now live in the namespace `hpx::parallel::execution`. The ols names can be still enabled at configure time by specifying `-DHPX_WITH_EXECUTION_POLICY_COMPATIBILITY=On` to CMake.
- The thread scheduling subsystem has undergone a major refactoring which results in significant performance improvements. We have also imroved the performance of creating `hpx::future` and of various facilities handling those.
- We have consolidated all of the code in HPX.Compute related to the integration of CUDA. `hpx::partitioned_vector` has been enabled to be usable with `hpx::compute::vector` which allows one to place the partitions on one or more GPU devices.
- Added new performance counters exposing various internals of the thread scheduling subsystem, such as the current idle- and busy-loop counters and instantaneous scheduler utilization.
- Extended and improved the use of the ITTNotify hooks allowing to collect performance counter data and function annotation information from within the Intel Amplifier tool.

Breaking changes

- We have dropped support for the gcc compiler versions V4.6 and 4.7. The minimal gcc version we now test on is gcc V4.8.
- We have removed (default) support for `boost::chrono` in interfaces, uses of it have been replaced with `std::chrono`. This facility can be still enabled at configure time by specifying `-DHPX_WITH_BOOST_CHRONO_COMPATIBILITY=On` to CMake.
- The parameter sequence for the `hpx::parallel::transform_reduce` overload taking one iterator range has changed to match the changes this algorithm has undergone while being moved to C++17.
- The algorithm `hpx::parallel::inner_product` has been renamed to `hpx::parallel::transform_reduce` to match the changes this algorithm has undergone while being moved to C++17.

[2342] https://github.com/STEllAR-GROUP/hpx/issues/2333
[2343] http://stellar-group.org/2016/09/vectorized-cpp-parallel-algorithms-with-hpx/

- the build options `HPX_WITH_COLOCATED_BACKWARDS_COMPATIBILITY` and `HPX_WITH_COMPONENT_GET_GID_COMPATIBILITY` are now disabled by default. Please change your code still depending on the deprecated interfaces.

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2596[2344] - Adding apex data
- PR #2595[2345] - Remove obsolete file
- Issue #2594[2346] - FindOpenCL.cmake mismatch with the official cmake module
- PR #2592[2347] - First attempt to introduce spmd_block in hpx
- Issue #2591[2348] - Feature request: continuation (then) which does not require the callable object to take a future<R> as parameter
- PR #2588[2349] - Daint fixes
- PR #2587[2350] - Fixing transfer_(continuation)_action::schedule
- PR #2585[2351] - Work around MSVC having an ICE when compiling with -Ob2
- PR #2583[2352] - changing 7zip command to 7za in roll_release.sh
- PR #2582[2353] - First attempt to introduce spmd_block in hpx
- PR #2581[2354] - Enable annotated function for parallel algorithms
- PR #2580[2355] - First attempt to introduce spmd_block in hpx
- PR #2579[2356] - Make thread NICE level setting an option
- PR #2578[2357] - Implementing enqueue instead of busy wait when no sender is available
- PR #2577[2358] - Retrieve -std=c++11 consistent nvcc flag
- PR #2576[2359] - Add missing dependencies of cuda based tests
- PR #2575[2360] - Remove warnings due to some captured variables
- PR #2573[2361] - Attempt to resolve resolve_locality
- PR #2572[2362] - Adding APEX hooks to background thread

2344 https://github.com/STEllAR-GROUP/hpx/pull/2596
2345 https://github.com/STEllAR-GROUP/hpx/pull/2595
2346 https://github.com/STEllAR-GROUP/hpx/issues/2594
2347 https://github.com/STEllAR-GROUP/hpx/pull/2592
2348 https://github.com/STEllAR-GROUP/hpx/issues/2591
2349 https://github.com/STEllAR-GROUP/hpx/pull/2588
2350 https://github.com/STEllAR-GROUP/hpx/pull/2587
2351 https://github.com/STEllAR-GROUP/hpx/pull/2585
2352 https://github.com/STEllAR-GROUP/hpx/pull/2583
2353 https://github.com/STEllAR-GROUP/hpx/pull/2582
2354 https://github.com/STEllAR-GROUP/hpx/pull/2581
2355 https://github.com/STEllAR-GROUP/hpx/pull/2580
2356 https://github.com/STEllAR-GROUP/hpx/pull/2579
2357 https://github.com/STEllAR-GROUP/hpx/pull/2578
2358 https://github.com/STEllAR-GROUP/hpx/pull/2577
2359 https://github.com/STEllAR-GROUP/hpx/pull/2576
2360 https://github.com/STEllAR-GROUP/hpx/pull/2575
2361 https://github.com/STEllAR-GROUP/hpx/pull/2573
2362 https://github.com/STEllAR-GROUP/hpx/pull/2572

- PR #2571[2363] - Pick up hpx.ignore_batch_env from config map
- PR #2570[2364] - Add commandline options –hpx:print-counters-locally
- PR #2569[2365] - Fix computeapi unit tests
- PR #2567[2366] - This adds another barrier::synchronize before registering performance counters
- PR #2564[2367] - Cray static toolchain support
- PR #2563[2368] - Fixed unhandled exception during startup
- PR #2562[2369] - Remove partitioned_vector.cu from build tree when nvcc is used
- Issue #2561[2370] - octo-tiger crash with commit 6e921495ff6c26f125d62629cbaad0525f14f7ab
- PR #2560[2371] - Prevent -Wundef warnings on Vc version checks
- PR #2559[2372] - Allowing CUDA callback to set the future directly from an OS thread
- PR #2558[2373] - Remove warnings due to float precisions
- PR #2557[2374] - Removing bogus handling of compile flags for CUDA
- PR #2556[2375] - Fixing scan partitioner
- PR #2554[2376] - Add more diagnostics to error thrown from find_appropriate_destination
- Issue #2555[2377] - No valid parcelport configured
- PR #2553[2378] - Add cmake cuda_arch option
- PR #2552[2379] - Remove incomplete datapar bindings to libflatarray
- PR #2551[2380] - Rename hwloc_topology to hwloc_topology_info
- PR #2550[2381] - Apex api updates
- PR #2549[2382] - Pre-include defines.hpp to get the macro HPX_HAVE_CUDA value
- PR #2548[2383] - Fixing issue with disconnect
- PR #2546[2384] - Some fixes around cuda clang partitioned_vector example
- PR #2545[2385] - Fix uses of the Vc2 datapar flags; the value, not the type, should be passed to functions

2363 https://github.com/STEllAR-GROUP/hpx/pull/2571
2364 https://github.com/STEllAR-GROUP/hpx/pull/2570
2365 https://github.com/STEllAR-GROUP/hpx/pull/2569
2366 https://github.com/STEllAR-GROUP/hpx/pull/2567
2367 https://github.com/STEllAR-GROUP/hpx/pull/2564
2368 https://github.com/STEllAR-GROUP/hpx/pull/2563
2369 https://github.com/STEllAR-GROUP/hpx/pull/2562
2370 https://github.com/STEllAR-GROUP/hpx/issues/2561
2371 https://github.com/STEllAR-GROUP/hpx/pull/2560
2372 https://github.com/STEllAR-GROUP/hpx/pull/2559
2373 https://github.com/STEllAR-GROUP/hpx/pull/2558
2374 https://github.com/STEllAR-GROUP/hpx/pull/2557
2375 https://github.com/STEllAR-GROUP/hpx/pull/2556
2376 https://github.com/STEllAR-GROUP/hpx/pull/2554
2377 https://github.com/STEllAR-GROUP/hpx/issues/2555
2378 https://github.com/STEllAR-GROUP/hpx/pull/2553
2379 https://github.com/STEllAR-GROUP/hpx/pull/2552
2380 https://github.com/STEllAR-GROUP/hpx/pull/2551
2381 https://github.com/STEllAR-GROUP/hpx/pull/2550
2382 https://github.com/STEllAR-GROUP/hpx/pull/2549
2383 https://github.com/STEllAR-GROUP/hpx/pull/2548
2384 https://github.com/STEllAR-GROUP/hpx/pull/2546
2385 https://github.com/STEllAR-GROUP/hpx/pull/2545

- PR #2542[2386] - Make HPX_WITH_MALLOC easier to use
- PR #2541[2387] - avoid recompiles when enabling/disabling examples
- PR #2540[2388] - Fixing usage of target_link_libraries()
- PR #2539[2389] - fix RPATH behaviour
- Issue #2538[2390] - HPX_WITH_CUDA corrupts compilation flags
- PR #2537[2391] - Add output of a Bazel Skylark extension for paths and compile options
- PR #2536[2392] - Add counter exposing total available memory to Windows as well
- PR #2535[2393] - Remove obsolete support for security
- Issue #2534[2394] - Remove command line option `--hpx:run-agas-server`
- PR #2533[2395] - Pre-cache locality endpoints during bootstrap
- PR #2532[2396] - Fixing handling of GIDs during serialization preprocessing
- PR #2531[2397] - Amend uses of the term "functor"
- PR #2529[2398] - added counter for reading available memory
- PR #2527[2399] - Facilities to create actions from lambdas
- PR #2526[2400] - Updated docs: HPX_WITH_EXAMPLES
- PR #2525[2401] - Remove warnings related to unused captured variables
- Issue #2524[2402] - CMAKE failed because it is missing: TCMALLOC_LIBRARY TCMALLOC_INCLUDE_DIR
- PR #2523[2403] - Fixing compose_cb stack overflow
- PR #2522[2404] - Instead of unlocking, ignore the lock while creating the message handler
- PR #2521[2405] - Create `LPROGRESS_` logging macro to simplify progress tracking and timings
- PR #2520[2406] - Intel 17 support
- PR #2519[2407] - Fix components example
- PR #2518[2408] - Fixing parcel scheduling

2386 https://github.com/STEllAR-GROUP/hpx/pull/2542
2387 https://github.com/STEllAR-GROUP/hpx/pull/2541
2388 https://github.com/STEllAR-GROUP/hpx/pull/2540
2389 https://github.com/STEllAR-GROUP/hpx/pull/2539
2390 https://github.com/STEllAR-GROUP/hpx/issues/2538
2391 https://github.com/STEllAR-GROUP/hpx/pull/2537
2392 https://github.com/STEllAR-GROUP/hpx/pull/2536
2393 https://github.com/STEllAR-GROUP/hpx/pull/2535
2394 https://github.com/STEllAR-GROUP/hpx/issues/2534
2395 https://github.com/STEllAR-GROUP/hpx/pull/2533
2396 https://github.com/STEllAR-GROUP/hpx/pull/2532
2397 https://github.com/STEllAR-GROUP/hpx/pull/2531
2398 https://github.com/STEllAR-GROUP/hpx/pull/2529
2399 https://github.com/STEllAR-GROUP/hpx/pull/2527
2400 https://github.com/STEllAR-GROUP/hpx/pull/2526
2401 https://github.com/STEllAR-GROUP/hpx/pull/2525
2402 https://github.com/STEllAR-GROUP/hpx/issues/2524
2403 https://github.com/STEllAR-GROUP/hpx/pull/2523
2404 https://github.com/STEllAR-GROUP/hpx/pull/2522
2405 https://github.com/STEllAR-GROUP/hpx/pull/2521
2406 https://github.com/STEllAR-GROUP/hpx/pull/2520
2407 https://github.com/STEllAR-GROUP/hpx/pull/2519
2408 https://github.com/STEllAR-GROUP/hpx/pull/2518

- Issue #2517[2409] - Race condition during Parcel Coalescing Handler creation
- Issue #2516[2410] - HPX locks up when using at least 256 localities
- Issue #2515[2411] - error: Install cannot find "/lib/hpx/libparcel_coalescing.so.0.9.99" but I can see that file
- PR #2514[2412] - Making sure that all continuations of a shared_future are invoked in order
- PR #2513[2413] - Fixing locks held during suspension
- PR #2512[2414] - MPI Parcelport improvements and fixes related to the background work changes
- PR #2511[2415] - Fixing bit-wise (zero-copy) serialization
- Issue #2509[2416] - Linking errors in hwloc_topology
- PR #2508[2417] - Added documentation for debugging with core files
- PR #2506[2418] - Fixing background work invocations
- PR #2505[2419] - Fix tuple serialization
- Issue #2504[2420] - Ensure continuations are called in the order they have been attached
- PR #2503[2421] - Adding serialization support for Vc v2 (datapar)
- PR #2502[2422] - Resolve various, minor compiler warnings
- PR #2501[2423] - Some other fixes around cuda examples
- Issue #2500[2424] - nvcc / cuda clang issue due to a missing -DHPX_WITH_CUDA flag
- PR #2499[2425] - Adding support for std::array to wait_all and friends
- PR #2498[2426] - Execute background work as HPX thread
- PR #2497[2427] - Fixing configuration options for spinlock-deadlock detection
- PR #2496[2428] - Accounting for different compilers in CrayKNL toolchain file
- PR #2494[2429] - Adding component base class which ties a component instance to a given executor
- PR #2493[2430] - Enable controlling amount of pending threads which must be available to allow thread stealing
- PR #2492[2431] - Adding new command line option –hpx:print-counter-reset

[2409] https://github.com/STEllAR-GROUP/hpx/issues/2517
[2410] https://github.com/STEllAR-GROUP/hpx/issues/2516
[2411] https://github.com/STEllAR-GROUP/hpx/issues/2515
[2412] https://github.com/STEllAR-GROUP/hpx/pull/2514
[2413] https://github.com/STEllAR-GROUP/hpx/pull/2513
[2414] https://github.com/STEllAR-GROUP/hpx/pull/2512
[2415] https://github.com/STEllAR-GROUP/hpx/pull/2511
[2416] https://github.com/STEllAR-GROUP/hpx/issues/2509
[2417] https://github.com/STEllAR-GROUP/hpx/pull/2508
[2418] https://github.com/STEllAR-GROUP/hpx/pull/2506
[2419] https://github.com/STEllAR-GROUP/hpx/pull/2505
[2420] https://github.com/STEllAR-GROUP/hpx/issues/2504
[2421] https://github.com/STEllAR-GROUP/hpx/pull/2503
[2422] https://github.com/STEllAR-GROUP/hpx/pull/2502
[2423] https://github.com/STEllAR-GROUP/hpx/pull/2501
[2424] https://github.com/STEllAR-GROUP/hpx/issues/2500
[2425] https://github.com/STEllAR-GROUP/hpx/pull/2499
[2426] https://github.com/STEllAR-GROUP/hpx/pull/2498
[2427] https://github.com/STEllAR-GROUP/hpx/pull/2497
[2428] https://github.com/STEllAR-GROUP/hpx/pull/2496
[2429] https://github.com/STEllAR-GROUP/hpx/pull/2494
[2430] https://github.com/STEllAR-GROUP/hpx/pull/2493
[2431] https://github.com/STEllAR-GROUP/hpx/pull/2492

- PR #2491[2432] - Resolve ambiguities when compiling with APEX
- PR #2490[2433] - Resuming threads waiting on future with higher priority
- Issue #2489[2434] - nvcc issue because -std=c++11 appears twice
- PR #2488[2435] - Adding performance counters exposing the internal idle and busy-loop counters
- PR #2487[2436] - Allowing for plain suspend to reschedule thread right away
- PR #2486[2437] - Only flag HPX code for CUDA if HPX_WITH_CUDA is set
- PR #2485[2438] - Making thread-queue parameters runtime-configurable
- PR #2484[2439] - Added atomic counter for parcel-destinations
- PR #2483[2440] - Added priority-queue lifo scheduler
- PR #2482[2441] - Changing scheduler to steal only if more than a minimal number of tasks are available
- PR #2481[2442] - Extending command line option –hpx:print-counter-destination to support value 'none'
- PR #2479[2443] - Added option to disable signal handler
- PR #2478[2444] - Making sure the sine performance counter module gets loaded only for the corresponding example
- Issue #2477[2445] - Breaking at a throw statement
- PR #2476[2446] - Annotated function
- PR #2475[2447] - Ensure that using %osthread% during logging will not throw for non-hpx threads
- PR #2474[2448] - Remove now superficial non_direct actions from base_lco and friends
- PR #2473[2449] - Refining support for ITTNotify
- PR #2472[2450] - Some fixes around hpx compute
- Issue #2470[2451] - redefinition of boost::detail::spinlock
- Issue #2469[2452] - Dataflow performance issue
- PR #2468[2453] - Perf docs update
- PR #2466[2454] - Guarantee to execute remote direct actions on HPX-thread

[2432] https://github.com/STEllAR-GROUP/hpx/pull/2491
[2433] https://github.com/STEllAR-GROUP/hpx/pull/2490
[2434] https://github.com/STEllAR-GROUP/hpx/issues/2489
[2435] https://github.com/STEllAR-GROUP/hpx/pull/2488
[2436] https://github.com/STEllAR-GROUP/hpx/pull/2487
[2437] https://github.com/STEllAR-GROUP/hpx/pull/2486
[2438] https://github.com/STEllAR-GROUP/hpx/pull/2485
[2439] https://github.com/STEllAR-GROUP/hpx/pull/2484
[2440] https://github.com/STEllAR-GROUP/hpx/pull/2483
[2441] https://github.com/STEllAR-GROUP/hpx/pull/2482
[2442] https://github.com/STEllAR-GROUP/hpx/pull/2481
[2443] https://github.com/STEllAR-GROUP/hpx/pull/2479
[2444] https://github.com/STEllAR-GROUP/hpx/pull/2478
[2445] https://github.com/STEllAR-GROUP/hpx/issues/2477
[2446] https://github.com/STEllAR-GROUP/hpx/pull/2476
[2447] https://github.com/STEllAR-GROUP/hpx/pull/2475
[2448] https://github.com/STEllAR-GROUP/hpx/pull/2474
[2449] https://github.com/STEllAR-GROUP/hpx/pull/2473
[2450] https://github.com/STEllAR-GROUP/hpx/pull/2472
[2451] https://github.com/STEllAR-GROUP/hpx/issues/2470
[2452] https://github.com/STEllAR-GROUP/hpx/issues/2469
[2453] https://github.com/STEllAR-GROUP/hpx/pull/2468
[2454] https://github.com/STEllAR-GROUP/hpx/pull/2466

- PR #2465[2455] - Improve demo : Async copy and fixed device handling
- PR #2464[2456] - Adding performance counter exposing instantaneous scheduler utilization
- PR #2463[2457] - Downcast to future<void>
- PR #2462[2458] - Fixed usage of ITT-Notify API with Intel Amplifier
- PR #2461[2459] - Cublas demo
- PR #2460[2460] - Fixing thread bindings
- PR #2459[2461] - Make -std=c++11 nvcc flag consistent for in-build and installed versions
- Issue #2457[2462] - Segmentation fault when registering a partitioned vector
- PR #2452[2463] - Properly releasing global barrier for unhandled exceptions
- PR #2451[2464] - Fixing long shutdown times
- PR #2450[2465] - Attempting to fix initialization errors on newer platforms (Boost V1.63)
- PR #2449[2466] - Replace BOOST_COMPILER_FENCE with an HPX version
- PR #2448[2467] - This fixes a possible race in the migration code
- **PR #2445[2468] - Fixing dataflow et.al. for futures or future-ranges wrapped** into ref()
- PR #2444[2469] - Fix segfaults
- PR #2443[2470] - Issue 2442
- Issue #2442[2471] - Mismatch between #if/#endif and namespace scope brackets in this_thread_executers.hpp
- Issue #2441[2472] - undeclared identifier BOOST_COMPILER_FENCE
- PR #2440[2473] - Knl build
- PR #2438[2474] - Datapar backend
- PR #2437[2475] - Adapt algorithm parameter sequence changes from C++17
- PR #2436[2476] - Adapt execution policy name changes from C++17
- Issue #2435[2477] - Trunk broken, undefined reference to hpx::thread::interrupt(hpx::thread::id, bool)

2455 https://github.com/STEllAR-GROUP/hpx/pull/2465
2456 https://github.com/STEllAR-GROUP/hpx/pull/2464
2457 https://github.com/STEllAR-GROUP/hpx/pull/2463
2458 https://github.com/STEllAR-GROUP/hpx/pull/2462
2459 https://github.com/STEllAR-GROUP/hpx/pull/2461
2460 https://github.com/STEllAR-GROUP/hpx/pull/2460
2461 https://github.com/STEllAR-GROUP/hpx/pull/2459
2462 https://github.com/STEllAR-GROUP/hpx/issues/2457
2463 https://github.com/STEllAR-GROUP/hpx/pull/2452
2464 https://github.com/STEllAR-GROUP/hpx/pull/2451
2465 https://github.com/STEllAR-GROUP/hpx/pull/2450
2466 https://github.com/STEllAR-GROUP/hpx/pull/2449
2467 https://github.com/STEllAR-GROUP/hpx/pull/2448
2468 https://github.com/STEllAR-GROUP/hpx/pull/2445
2469 https://github.com/STEllAR-GROUP/hpx/pull/2444
2470 https://github.com/STEllAR-GROUP/hpx/pull/2443
2471 https://github.com/STEllAR-GROUP/hpx/issues/2442
2472 https://github.com/STEllAR-GROUP/hpx/issues/2441
2473 https://github.com/STEllAR-GROUP/hpx/pull/2440
2474 https://github.com/STEllAR-GROUP/hpx/pull/2438
2475 https://github.com/STEllAR-GROUP/hpx/pull/2437
2476 https://github.com/STEllAR-GROUP/hpx/pull/2436
2477 https://github.com/STEllAR-GROUP/hpx/issues/2435

- PR #2434[2478] - More fixes to resource manager
- PR #2433[2479] - Added versions of `hpx::get_ptr` taking client side representations
- PR #2432[2480] - Warning fixes
- PR #2431[2481] - Adding facility representing set of performance counters
- PR #2430[2482] - Fix parallel_executor thread spawning
- PR #2429[2483] - Fix attribute warning for gcc
- Issue #2427[2484] - Seg fault running octo-tiger with latest HPX commit
- Issue #2426[2485] - Bug in 9592f5c0bc29806fce0dbe73f35b6ca7e027edcb causes immediate crash in Octo-tiger
- PR #2425[2486] - Fix nvcc errors due to constexpr specifier
- Issue #2424[2487] - Async action on component present on hpx::find_here is executing synchronously
- PR #2423[2488] - Fix nvcc errors due to constexpr specifier
- PR #2422[2489] - Implementing hpx::this_thread thread data functions
- PR #2421[2490] - Adding benchmark for wait_all
- Issue #2420[2491] - Returning object of a component client from another component action fails
- PR #2419[2492] - Infiniband parcelport
- Issue #2418[2493] - gcc + nvcc fails to compile code that uses partitioned_vector
- PR #2417[2494] - Fixing context switching
- PR #2416[2495] - Adding fixes and workarounds to allow compilation with nvcc/msvc (VS2015up3)
- PR #2415[2496] - Fix errors coming from hpx compute examples
- PR #2414[2497] - Fixing msvc12
- PR #2413[2498] - Enable cuda/nvcc or cuda/clang when using add_hpx_executable()
- PR #2412[2499] - Fix issue in HPX_SetupTarget.cmake when cuda is used
- PR #2411[2500] - This fixes the core compilation issues with MSVC12

2478 https://github.com/STEllAR-GROUP/hpx/pull/2434
2479 https://github.com/STEllAR-GROUP/hpx/pull/2433
2480 https://github.com/STEllAR-GROUP/hpx/pull/2432
2481 https://github.com/STEllAR-GROUP/hpx/pull/2431
2482 https://github.com/STEllAR-GROUP/hpx/pull/2430
2483 https://github.com/STEllAR-GROUP/hpx/pull/2429
2484 https://github.com/STEllAR-GROUP/hpx/issues/2427
2485 https://github.com/STEllAR-GROUP/hpx/issues/2426
2486 https://github.com/STEllAR-GROUP/hpx/pull/2425
2487 https://github.com/STEllAR-GROUP/hpx/issues/2424
2488 https://github.com/STEllAR-GROUP/hpx/pull/2423
2489 https://github.com/STEllAR-GROUP/hpx/pull/2422
2490 https://github.com/STEllAR-GROUP/hpx/pull/2421
2491 https://github.com/STEllAR-GROUP/hpx/issues/2420
2492 https://github.com/STEllAR-GROUP/hpx/pull/2419
2493 https://github.com/STEllAR-GROUP/hpx/issues/2418
2494 https://github.com/STEllAR-GROUP/hpx/pull/2417
2495 https://github.com/STEllAR-GROUP/hpx/pull/2416
2496 https://github.com/STEllAR-GROUP/hpx/pull/2415
2497 https://github.com/STEllAR-GROUP/hpx/pull/2414
2498 https://github.com/STEllAR-GROUP/hpx/pull/2413
2499 https://github.com/STEllAR-GROUP/hpx/pull/2412
2500 https://github.com/STEllAR-GROUP/hpx/pull/2411

- Issue #2410[2501] - `undefined reference to opal_hwloc191_hwloc_.....`
- PR #2409[2502] - Fixing locking for channel and receive_buffer
- PR #2407[2503] - Solving #2402 and #2403
- PR #2406[2504] - Improve guards
- PR #2405[2505] - Enable parallel::for_each for iterators returning proxy types
- PR #2404[2506] - Forward the explicitly given result_type in the hpx invoke
- Issue #2403[2507] - datapar_execution + zip iterator: lambda arguments aren't references
- Issue #2402[2508] - datapar algorithm instantiated with wrong type #2402
- PR #2401[2509] - Added support for imported libraries to HPX_Libraries.cmake
- PR #2400[2510] - Use CMake policy CMP0060
- Issue #2399[2511] - Error trying to push back vector of futures to vector
- PR #2398[2512] - Allow config #defines to be written out to custom config/defines.hpp
- Issue #2397[2513] - CMake generated config defines can cause tedious rebuilds category
- Issue #2396[2514] - BOOST_ROOT paths are not used at link time
- PR #2395[2515] - Fix target_link_libraries() issue when HPX Cuda is enabled
- Issue #2394[2516] - Template compilation error using HPX_WITH_DATAPAR_LIBFLATARRAY
- PR #2393[2517] - Fixing lock registration for recursive mutex
- PR #2392[2518] - Add keywords in target_link_libraries in hpx_setup_target
- PR #2391[2519] - Clang goroutines
- Issue #2390[2520] - Adapt execution policy name changes from C++17
- PR #2389[2521] - Chunk allocator and pool are not used and are obsolete
- PR #2388[2522] - Adding functionalities to datapar needed by octotiger
- PR #2387[2523] - Fixing race condition for early parcels

2501 https://github.com/STEllAR-GROUP/hpx/issues/2410
2502 https://github.com/STEllAR-GROUP/hpx/pull/2409
2503 https://github.com/STEllAR-GROUP/hpx/pull/2407
2504 https://github.com/STEllAR-GROUP/hpx/pull/2406
2505 https://github.com/STEllAR-GROUP/hpx/pull/2405
2506 https://github.com/STEllAR-GROUP/hpx/pull/2404
2507 https://github.com/STEllAR-GROUP/hpx/issues/2403
2508 https://github.com/STEllAR-GROUP/hpx/issues/2402
2509 https://github.com/STEllAR-GROUP/hpx/pull/2401
2510 https://github.com/STEllAR-GROUP/hpx/pull/2400
2511 https://github.com/STEllAR-GROUP/hpx/issues/2399
2512 https://github.com/STEllAR-GROUP/hpx/pull/2398
2513 https://github.com/STEllAR-GROUP/hpx/issues/2397
2514 https://github.com/STEllAR-GROUP/hpx/issues/2396
2515 https://github.com/STEllAR-GROUP/hpx/pull/2395
2516 https://github.com/STEllAR-GROUP/hpx/issues/2394
2517 https://github.com/STEllAR-GROUP/hpx/pull/2393
2518 https://github.com/STEllAR-GROUP/hpx/pull/2392
2519 https://github.com/STEllAR-GROUP/hpx/pull/2391
2520 https://github.com/STEllAR-GROUP/hpx/issues/2390
2521 https://github.com/STEllAR-GROUP/hpx/pull/2389
2522 https://github.com/STEllAR-GROUP/hpx/pull/2388
2523 https://github.com/STEllAR-GROUP/hpx/pull/2387

- Issue #2386[2524] - Lock registration broken for recursive_mutex
- PR #2385[2525] - Datapar zip iterator
- PR #2384[2526] - Fixing race condition in for_loop_reduction
- PR #2383[2527] - Continuations
- PR #2382[2528] - add LibFlatArray-based backend for datapar
- PR #2381[2529] - remove unused typedef to get rid of compiler warnings
- PR #2380[2530] - Tau cleanup
- PR #2379[2531] - Can send immediate
- PR #2378[2532] - Renaming copy_helper/copy_n_helper/move_helper/move_n_helper
- Issue #2376[2533] - Boost trunk's spinlock initializer fails to compile
- PR #2375[2534] - Add support for minimal thread local data
- PR #2374[2535] - Adding API functions set_config_entry_callback
- PR #2373[2536] - Add a simple utility for debugging that gives supended task backtraces
- PR #2372[2537] - Barrier Fixes
- Issue #2370[2538] - Can't wait on a wrapped future
- PR #2369[2539] - Fixing stable_partition
- PR #2367[2540] - Fixing find_prefixes for Windows platforms
- PR #2366[2541] - Testing for experimental/optional only in C++14 mode
- PR #2364[2542] - Adding set_config_entry
- PR #2363[2543] - Fix papi
- PR #2362[2544] - Adding missing macros for new non-direct actions
- PR #2361[2545] - Improve cmake output to help debug compiler incompatibility check
- PR #2360[2546] - Fixing race condition in condition_variable

2524 https://github.com/STEllAR-GROUP/hpx/issues/2386
2525 https://github.com/STEllAR-GROUP/hpx/pull/2385
2526 https://github.com/STEllAR-GROUP/hpx/pull/2384
2527 https://github.com/STEllAR-GROUP/hpx/pull/2383
2528 https://github.com/STEllAR-GROUP/hpx/pull/2382
2529 https://github.com/STEllAR-GROUP/hpx/pull/2381
2530 https://github.com/STEllAR-GROUP/hpx/pull/2380
2531 https://github.com/STEllAR-GROUP/hpx/pull/2379
2532 https://github.com/STEllAR-GROUP/hpx/pull/2378
2533 https://github.com/STEllAR-GROUP/hpx/issues/2376
2534 https://github.com/STEllAR-GROUP/hpx/pull/2375
2535 https://github.com/STEllAR-GROUP/hpx/pull/2374
2536 https://github.com/STEllAR-GROUP/hpx/pull/2373
2537 https://github.com/STEllAR-GROUP/hpx/pull/2372
2538 https://github.com/STEllAR-GROUP/hpx/issues/2370
2539 https://github.com/STEllAR-GROUP/hpx/pull/2369
2540 https://github.com/STEllAR-GROUP/hpx/pull/2367
2541 https://github.com/STEllAR-GROUP/hpx/pull/2366
2542 https://github.com/STEllAR-GROUP/hpx/pull/2364
2543 https://github.com/STEllAR-GROUP/hpx/pull/2363
2544 https://github.com/STEllAR-GROUP/hpx/pull/2362
2545 https://github.com/STEllAR-GROUP/hpx/pull/2361
2546 https://github.com/STEllAR-GROUP/hpx/pull/2360

- PR #2359[2547] - Fixing shutdown when parcels are still in flight
- Issue #2357[2548] - failed to insert console_print_action into typename_to_id_t registry
- PR #2356[2549] - Fixing return type of get_iterator_tuple
- PR #2355[2550] - Fixing compilation against Boost 1 62
- PR #2354[2551] - Adding serialization for mask_type if CPU_COUNT > 64
- PR #2353[2552] - Adding hooks to tie in APEX into the parcel layer
- Issue #2352[2553] - Compile errors when using intel 17 beta (for KNL) on edison
- PR #2351[2554] - Fix function vtable get_function_address implementation
- Issue #2350[2555] - Build failure - master branch (4de09f5) with Intel Compiler v17
- PR #2349[2556] - Enabling zero-copy serialization support for std::vector<>
- PR #2348[2557] - Adding test to verify #2334 is fixed
- PR #2347[2558] - Bug fixes for hpx.compute and hpx::lcos::channel
- PR #2346[2559] - Removing cmake "find" files that are in the APEX cmake Modules
- PR #2345[2560] - Implemented parallel::stable_partition
- PR #2344[2561] - Making hpx::lcos::channel usable with basename registration
- PR #2343[2562] - Fix a couple of examples that failed to compile after recent api changes
- Issue #2342[2563] - Enabling APEX causes link errors
- PR #2341[2564] - Removing cmake "find" files that are in the APEX cmake Modules
- PR #2340[2565] - Implemented all existing datapar algorithms using Boost.SIMD
- PR #2339[2566] - Fixing 2338
- PR #2338[2567] - Possible race in sliding semaphore
- PR #2337[2568] - Adjust osu_latency test to measure window_size parcels in flight at once
- PR #2336[2569] - Allowing remote direct actions to be executed without spawning a task

[2547] https://github.com/STEllAR-GROUP/hpx/pull/2359
[2548] https://github.com/STEllAR-GROUP/hpx/issues/2357
[2549] https://github.com/STEllAR-GROUP/hpx/pull/2356
[2550] https://github.com/STEllAR-GROUP/hpx/pull/2355
[2551] https://github.com/STEllAR-GROUP/hpx/pull/2354
[2552] https://github.com/STEllAR-GROUP/hpx/pull/2353
[2553] https://github.com/STEllAR-GROUP/hpx/issues/2352
[2554] https://github.com/STEllAR-GROUP/hpx/pull/2351
[2555] https://github.com/STEllAR-GROUP/hpx/issues/2350
[2556] https://github.com/STEllAR-GROUP/hpx/pull/2349
[2557] https://github.com/STEllAR-GROUP/hpx/pull/2348
[2558] https://github.com/STEllAR-GROUP/hpx/pull/2347
[2559] https://github.com/STEllAR-GROUP/hpx/pull/2346
[2560] https://github.com/STEllAR-GROUP/hpx/pull/2345
[2561] https://github.com/STEllAR-GROUP/hpx/pull/2344
[2562] https://github.com/STEllAR-GROUP/hpx/pull/2343
[2563] https://github.com/STEllAR-GROUP/hpx/issues/2342
[2564] https://github.com/STEllAR-GROUP/hpx/pull/2341
[2565] https://github.com/STEllAR-GROUP/hpx/pull/2340
[2566] https://github.com/STEllAR-GROUP/hpx/pull/2339
[2567] https://github.com/STEllAR-GROUP/hpx/pull/2338
[2568] https://github.com/STEllAR-GROUP/hpx/pull/2337
[2569] https://github.com/STEllAR-GROUP/hpx/pull/2336

- PR #2335[2570] - Making sure multiple components are properly initialized from arguments
- Issue #2334[2571] - Cannot construct component with large vector on a remote locality
- PR #2332[2572] - Fixing hpx::lcos::local::barrier
- PR #2331[2573] - Updating APEX support to include OTF2
- PR #2330[2574] - Support for data-parallelism for parallel algorithms
- Issue #2329[2575] - Coordinate settings in cmake
- PR #2328[2576] - fix LibGeoDecomp builds with HPX + GCC 5.3.0 + CUDA 8RC
- PR #2326[2577] - Making scan_partitioner work (for now)
- Issue #2323[2578] - Constructing a vector of components only correctly initializes the first component
- PR #2322[2579] - Fix problems that bubbled up after merging #2278
- PR #2321[2580] - Scalable barrier
- PR #2320[2581] - Std flag fixes
- Issue #2319[2582] - -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[2583] - Improve handling of –hpx:bind=<bind-spec>
- PR #2317[2584] - Making sure command line warnings are printed once only
- PR #2316[2585] - Fixing command line handling for default bind mode
- PR #2315[2586] - Set id_retrieved if set_id is present
- Issue #2314[2587] - Warning for requested/allocated thread discrepancy is printed twice
- Issue #2313[2588] - –hpx:print-bind doesn't work with –hpx:pu-step
- Issue #2312[2589] - –hpx:bind range specifier restrictions are overly restrictive
- Issue #2311[2590] - hpx_0.9.99 out of project build fails
- PR #2310[2591] - Simplify function registration
- PR #2309[2592] - Spelling and grammar revisions in documentation (and some code)

[2570] https://github.com/STEllAR-GROUP/hpx/pull/2335
[2571] https://github.com/STEllAR-GROUP/hpx/issues/2334
[2572] https://github.com/STEllAR-GROUP/hpx/pull/2332
[2573] https://github.com/STEllAR-GROUP/hpx/pull/2331
[2574] https://github.com/STEllAR-GROUP/hpx/pull/2330
[2575] https://github.com/STEllAR-GROUP/hpx/issues/2329
[2576] https://github.com/STEllAR-GROUP/hpx/pull/2328
[2577] https://github.com/STEllAR-GROUP/hpx/pull/2326
[2578] https://github.com/STEllAR-GROUP/hpx/issues/2323
[2579] https://github.com/STEllAR-GROUP/hpx/pull/2322
[2580] https://github.com/STEllAR-GROUP/hpx/pull/2321
[2581] https://github.com/STEllAR-GROUP/hpx/pull/2320
[2582] https://github.com/STEllAR-GROUP/hpx/issues/2319
[2583] https://github.com/STEllAR-GROUP/hpx/pull/2318
[2584] https://github.com/STEllAR-GROUP/hpx/pull/2317
[2585] https://github.com/STEllAR-GROUP/hpx/pull/2316
[2586] https://github.com/STEllAR-GROUP/hpx/pull/2315
[2587] https://github.com/STEllAR-GROUP/hpx/issues/2314
[2588] https://github.com/STEllAR-GROUP/hpx/issues/2313
[2589] https://github.com/STEllAR-GROUP/hpx/issues/2312
[2590] https://github.com/STEllAR-GROUP/hpx/issues/2311
[2591] https://github.com/STEllAR-GROUP/hpx/pull/2310
[2592] https://github.com/STEllAR-GROUP/hpx/pull/2309

- PR #2306[2593] - Correct minor typo in the documentation
- PR #2305[2594] - Cleaning up and fixing parcel coalescing
- PR #2304[2595] - Inspect checks for stream related includes
- PR #2303[2596] - Add functionality allowing to enumerate threads of given state
- PR #2301[2597] - Algorithm overloads fix for VS2013
- PR #2300[2598] - Use <cstdint>, add inspect checks
- PR #2299[2599] - Replace boost::[c]ref with std::[c]ref, add inspect checks
- PR #2297[2600] - Fixing compilation with no hw_loc
- PR #2296[2601] - Hpx compute
- PR #2295[2602] - Making sure for_loop(execution::par, 0, N, . . .) is actually executed in parallel
- PR #2294[2603] - Throwing exceptions if the runtime is not up and running
- PR #2293[2604] - Removing unused parcel port code
- PR #2292[2605] - Refactor function vtables
- PR #2291[2606] - Fixing 2286
- PR #2290[2607] - Simplify algorithm overloads
- PR #2289[2608] - Adding performance counters reporting parcel related data on a per-action basis
- Issue #2288[2609] - Remove dormant parcelports
- Issue #2286[2610] - adjustments to parcel handling to support parcelports that do not need a connection cache
- PR #2285[2611] - add CMake option to disable package export
- PR #2283[2612] - Add more inspect checks for use of deprecated components
- Issue #2282[2613] - Arithmetic exception in executor static chunker
- Issue #2281[2614] - For loop doesn't parallelize
- PR #2280[2615] - Fixing 2277: build failure with PAPI

[2593] https://github.com/STEllAR-GROUP/hpx/pull/2306
[2594] https://github.com/STEllAR-GROUP/hpx/pull/2305
[2595] https://github.com/STEllAR-GROUP/hpx/pull/2304
[2596] https://github.com/STEllAR-GROUP/hpx/pull/2303
[2597] https://github.com/STEllAR-GROUP/hpx/pull/2301
[2598] https://github.com/STEllAR-GROUP/hpx/pull/2300
[2599] https://github.com/STEllAR-GROUP/hpx/pull/2299
[2600] https://github.com/STEllAR-GROUP/hpx/pull/2297
[2601] https://github.com/STEllAR-GROUP/hpx/pull/2296
[2602] https://github.com/STEllAR-GROUP/hpx/pull/2295
[2603] https://github.com/STEllAR-GROUP/hpx/pull/2294
[2604] https://github.com/STEllAR-GROUP/hpx/pull/2293
[2605] https://github.com/STEllAR-GROUP/hpx/pull/2292
[2606] https://github.com/STEllAR-GROUP/hpx/pull/2291
[2607] https://github.com/STEllAR-GROUP/hpx/pull/2290
[2608] https://github.com/STEllAR-GROUP/hpx/pull/2289
[2609] https://github.com/STEllAR-GROUP/hpx/issues/2288
[2610] https://github.com/STEllAR-GROUP/hpx/issues/2286
[2611] https://github.com/STEllAR-GROUP/hpx/pull/2285
[2612] https://github.com/STEllAR-GROUP/hpx/pull/2283
[2613] https://github.com/STEllAR-GROUP/hpx/issues/2282
[2614] https://github.com/STEllAR-GROUP/hpx/issues/2281
[2615] https://github.com/STEllAR-GROUP/hpx/pull/2280

- PR #2279[2616] - Child vs parent stealing
- Issue #2277[2617] - master branch build failure (53c5b4f) with papi
- PR #2276[2618] - Compile time launch policies
- PR #2275[2619] - Replace boost::chrono with std::chrono in interfaces
- PR #2274[2620] - Replace most uses of Boost.Assign with initializer list
- PR #2273[2621] - Fixed typos
- PR #2272[2622] - Inspect checks
- PR #2270[2623] - Adding test verifying -Ihpx.os_threads=all
- PR #2269[2624] - Added inspect check for now obsolete boost type traits
- PR #2268[2625] - Moving more code into source files
- Issue #2267[2626] - Add inspect support to deprecate Boost.TypeTraits
- PR #2265[2627] - Adding channel LCO
- PR #2264[2628] - Make support for std::ref mandatory
- PR #2263[2629] - Constrain tuple_member forwarding constructor
- Issue #2262[2630] - Test hpx.os_threads=all
- Issue #2261[2631] - OS X: Error: no matching constructor for initialization of 'hpx::lcos::local::condition_variable_any'
- Issue #2260[2632] - Make support for std::ref mandatory
- PR #2259[2633] - Remove most of Boost.MPL, Boost.EnableIf and Boost.TypeTraits
- PR #2258[2634] - Fixing #2256
- PR #2257[2635] - Fixing launch process
- Issue #2256[2636] - Actions are not registered if not invoked
- PR #2255[2637] - Coalescing histogram
- PR #2254[2638] - Silence explicit initialization in copy-constructor warnings

2616 https://github.com/STEllAR-GROUP/hpx/pull/2279
2617 https://github.com/STEllAR-GROUP/hpx/issues/2277
2618 https://github.com/STEllAR-GROUP/hpx/pull/2276
2619 https://github.com/STEllAR-GROUP/hpx/pull/2275
2620 https://github.com/STEllAR-GROUP/hpx/pull/2274
2621 https://github.com/STEllAR-GROUP/hpx/pull/2273
2622 https://github.com/STEllAR-GROUP/hpx/pull/2272
2623 https://github.com/STEllAR-GROUP/hpx/pull/2270
2624 https://github.com/STEllAR-GROUP/hpx/pull/2269
2625 https://github.com/STEllAR-GROUP/hpx/pull/2268
2626 https://github.com/STEllAR-GROUP/hpx/issues/2267
2627 https://github.com/STEllAR-GROUP/hpx/pull/2265
2628 https://github.com/STEllAR-GROUP/hpx/pull/2264
2629 https://github.com/STEllAR-GROUP/hpx/pull/2263
2630 https://github.com/STEllAR-GROUP/hpx/issues/2262
2631 https://github.com/STEllAR-GROUP/hpx/issues/2261
2632 https://github.com/STEllAR-GROUP/hpx/issues/2260
2633 https://github.com/STEllAR-GROUP/hpx/pull/2259
2634 https://github.com/STEllAR-GROUP/hpx/pull/2258
2635 https://github.com/STEllAR-GROUP/hpx/pull/2257
2636 https://github.com/STEllAR-GROUP/hpx/issues/2256
2637 https://github.com/STEllAR-GROUP/hpx/pull/2255
2638 https://github.com/STEllAR-GROUP/hpx/pull/2254

- PR #2253[2639] - Drop support for GCC 4.6 and 4.7
- PR #2252[2640] - Prepare V1.0
- PR #2251[2641] - Convert to 0.9.99
- PR #2249[2642] - Adding iterator_facade and iterator_adaptor
- Issue #2248[2643] - Need a feature to yield to a new task immediately
- PR #2246[2644] - Adding split_future
- PR #2245[2645] - Add an example for handing over a component instance to a dynamically launched locality
- Issue #2243[2646] - Add example demonstrating AGAS symbolic name registration
- Issue #2242[2647] - pkgconfig test broken on CentOS 7 / Boost 1.61
- Issue #2241[2648] - Compilation error for partitioned vector in hpx_compute branch
- PR #2240[2649] - Fixing termination detection on one locality
- Issue #2239[2650] - Create a new facility lcos::split_all
- Issue #2236[2651] - hpx::cout vs. std::cout
- PR #2232[2652] - Implement local-only primary namespace service
- Issue #2147[2653] - would like to know how much data is being routed by particular actions
- Issue #2109[2654] - Warning while compiling hpx
- Issue #1973[2655] - Setting INTERFACE_COMPILE_OPTIONS for hpx_init in CMake taints Fortran_FLAGS
- Issue #1864[2656] - run_guarded using bound function ignores reference
- Issue #1754[2657] - Running with TCP parcelport causes immediate crash or freeze
- Issue #1655[2658] - Enable zip_iterator to be used with Boost traversal iterator categories
- Issue #1591[2659] - Optimize AGAS for shared memory only operation
- Issue #1401[2660] - Need an efficient infiniband parcelport
- Issue #1125[2661] - Fix the IPC parcelport

[2639] https://github.com/STEllAR-GROUP/hpx/pull/2253
[2640] https://github.com/STEllAR-GROUP/hpx/pull/2252
[2641] https://github.com/STEllAR-GROUP/hpx/pull/2251
[2642] https://github.com/STEllAR-GROUP/hpx/pull/2249
[2643] https://github.com/STEllAR-GROUP/hpx/issues/2248
[2644] https://github.com/STEllAR-GROUP/hpx/pull/2246
[2645] https://github.com/STEllAR-GROUP/hpx/pull/2245
[2646] https://github.com/STEllAR-GROUP/hpx/issues/2243
[2647] https://github.com/STEllAR-GROUP/hpx/issues/2242
[2648] https://github.com/STEllAR-GROUP/hpx/issues/2241
[2649] https://github.com/STEllAR-GROUP/hpx/pull/2240
[2650] https://github.com/STEllAR-GROUP/hpx/issues/2239
[2651] https://github.com/STEllAR-GROUP/hpx/issues/2236
[2652] https://github.com/STEllAR-GROUP/hpx/pull/2232
[2653] https://github.com/STEllAR-GROUP/hpx/issues/2147
[2654] https://github.com/STEllAR-GROUP/hpx/issues/2109
[2655] https://github.com/STEllAR-GROUP/hpx/issues/1973
[2656] https://github.com/STEllAR-GROUP/hpx/issues/1864
[2657] https://github.com/STEllAR-GROUP/hpx/issues/1754
[2658] https://github.com/STEllAR-GROUP/hpx/issues/1655
[2659] https://github.com/STEllAR-GROUP/hpx/issues/1591
[2660] https://github.com/STEllAR-GROUP/hpx/issues/1401
[2661] https://github.com/STEllAR-GROUP/hpx/issues/1125

- Issue #839[2662] - Refactor ibverbs and shmem parcelport
- Issue #702[2663] - Add instrumentation of parcel layer
- Issue #668[2664] - Implement ispc task interface
- Issue #533[2665] - Thread queue/deque internal parameters should be runtime configurable
- Issue #475[2666] - Create a means of combining performance counters into querysets

2.11.10 *HPX* V0.9.99 (Jul 15, 2016)

General changes

As the version number of this release hints, we consider this release to be a preview for the upcoming *HPX* V1.0. All of the functionalities we set out to implement for V1.0 are in place; all of the features we wanted to have exposed are ready. We are very happy with the stability and performance of *HPX* and we would like to present this release to the community in order for us to gather broad feedback before releasing V1.0. We still expect for some minor details to change, but on the whole this release represents what we would like to have in a V1.0.

Overall, since the last release we have had almost 1600 commits while closing almost 400 tickets. These numbers reflect the incredible development activity we have seen over the last couple of months. We would like to express a big 'Thank you!' to all contributors and those who helped to make this release happen.

The most notable addition in terms of new functionality available with this release is the full implementation of object migration (i.e. the ability to transparently move *HPX* components to a different compute node). Additionally, this release of *HPX* cleans up many minor issues and some API inconsistencies.

Here are some of the main highlights and changes for this release (in no particular order):

- We have fixed a couple of issues in AGAS and the parcel layer which have caused hangs, segmentation faults at exit, and a slowdown of applications over time. Fixing those has significantly increased the overall stability and performance of distributed runs.
- We have started to add parallel algorithm overloads based on the C++ Extensions for Ranges (N4560[2667]) proposal. This also includes the addition of projections to the existing algorithms. Please see Issue #1668[2668] for a list of algorithms which have been adapted to N4560[2669].
- We have implemented index-based parallel for-loops based on a corresponding standardization proposal (P0075R1[2670]). Please see Issue #2016[2671] 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[2672] 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.

[2662] https://github.com/STEllAR-GROUP/hpx/issues/839
[2663] https://github.com/STEllAR-GROUP/hpx/issues/702
[2664] https://github.com/STEllAR-GROUP/hpx/issues/668
[2665] https://github.com/STEllAR-GROUP/hpx/issues/533
[2666] https://github.com/STEllAR-GROUP/hpx/issues/475
[2667] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf
[2668] https://github.com/STEllAR-GROUP/hpx/issues/1668
[2669] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf
[2670] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0075r1.pdf
[2671] https://github.com/STEllAR-GROUP/hpx/issues/2016
[2672] https://github.com/STEllAR-GROUP/hpx/issues/1141

- Added `hpx::make_future<R>(future<T> &&)` allowing to convert a future of any type `T` into a future of any other type `R`, either based on default conversion rules of the embedded types or using a given explicit conversion function.
- We finally finished the implementation of transparent migration of components to another locality. It is now possible to trigger a migration operation without 'stopping the world' for the object to migrate. *HPX* will make sure that no work is being performed on an object before it is migrated and that all subsequently scheduled work for the migrated object will be transparently forwarded to the new locality. Please note that the global id of the migrated object does not change, thus the application will not have to be changed in any way to support this new functionality. Please note that this feature is currently considered experimental. See Issue #559[2673] and PR #1966[2674] 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[2675] and PR #1912[2676] for more information.
- Adding overloads of `gather_here` and `gather_there` to accept the plain values of the data to gather (in addition to the existing overloads expecting futures).
- We have cleaned up and refactored large parts of the code base. This helped reducing compile and link times of *HPX* itself and also of applications depending on it. We have further decreased the dependency of *HPX* on the Boost libraries by replacing part of those with facilities available from the standard libraries.
- Wherever possible we have removed dependencies of our API on Boost by replacing those with the equivalent facility from the C++11 standard library.
- We have added new performance counters for parcel coalescing, file-IO, the AGAS cache, and overall scheduler time. Resetting performance counters has been overhauled and fixed.
- We have introduced a generic client type `hpx::components::client<>` and added support for using it with `hpx::async`. This removes the necessity to implement specific client types for every component type without losing type safety. This deemphasizes the need for using the low level `hpx::id_type` for referencing (possibly remote) component instances. The plan is to deprecate the direct use of `hpx::id_type` in user code in the future.
- We have added a special iterator which supports automatic prefetching of one or more arrays for speeding up loop-like code (see `hpx::parallel::util::make_prefetcher_context()`).
- We have extended the interfaces exposed from executors (as proposed by N4406[2677]) 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[2678] is defined at configuration time. Please explicitly qualify all uses of the dataflow facility if you enable this compatibility setting and encounter ambiguities.

[2673] https://github.com/STEllAR-GROUP/hpx/issues/559
[2674] https://github.com/STEllAR-GROUP/hpx/pull/1966
[2675] https://github.com/STEllAR-GROUP/hpx/issues/1265
[2676] https://github.com/STEllAR-GROUP/hpx/pull/1912
[2677] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf
[2678] https://www.cmake.org

- The adaptation of the C++ Extensions for Ranges (N4560[2679]) proposal imposes some breaking changes related to the return types of some of the parallel algorithms. Please see Issue #1668[2680] for a list of algorithms which have already been adapted.
- The facility `hpx::lcos::make_future_void()` has been replaced by `hpx::make_future<void>()`.
- We have removed support for Intel V13 and gcc 4.4.x.
- We have removed (default) support for the generic `hpx::parallel::execution_poliy` because it was removed from the Parallelism TS (__cpp11_n4104__) while it was being added to the upcoming C++17 Standard. This facility can be still enabled at configure time by specifying `-DHPX_WITH_GENERIC_EXECUTION_POLICY=On` to CMake.
- Uses of `boost::shared_ptr` and related facilities have been replaced with `std::shared_ptr` and friends. Uses of `boost::unique_lock`, `boost::lock_guard` etc. have also been replaced by the equivalent (and equally named) tools available from the C++11 standard library.
- Facilities that used to expect an explicit `boost::unique_lock` now take an `std::unique_lock`. Additionally, `condition_variable` no longer aliases `condition_variable_any`; its interface now only works with `std::unique_lock<local::mutex>`.
- Uses of `boost::function`, `boost::bind`, `boost::tuple` have been replaced by the corresponding facilities in *HPX* (`hpx::util::function`, `hpx::util::bind`, and `hpx::util::tuple`, respectively).

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- PR #2250[2681] - change default chunker of parallel executor to static one
- PR #2247[2682] - HPX on ppc64le
- PR #2244[2683] - Fixing MSVC problems
- PR #2238[2684] - Fixing small typos
- PR #2237[2685] - Fixing small typos
- PR #2234[2686] - Fix broken add test macro when extra args are passed in
- PR #2231[2687] - Fixing possible race during future awaiting in serialization
- PR #2230[2688] - Fix stream nvcc
- PR #2229[2689] - Fixed run_as_hpx_thread
- PR #2228[2690] - On prefetching_test branch : adding prefetching_iterator and related tests used for prefetching containers within lambda functions

[2679] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4560.pdf
[2680] https://github.com/STEllAR-GROUP/hpx/issues/1668
[2681] https://github.com/STEllAR-GROUP/hpx/pull/2250
[2682] https://github.com/STEllAR-GROUP/hpx/pull/2247
[2683] https://github.com/STEllAR-GROUP/hpx/pull/2244
[2684] https://github.com/STEllAR-GROUP/hpx/pull/2238
[2685] https://github.com/STEllAR-GROUP/hpx/pull/2237
[2686] https://github.com/STEllAR-GROUP/hpx/pull/2234
[2687] https://github.com/STEllAR-GROUP/hpx/pull/2231
[2688] https://github.com/STEllAR-GROUP/hpx/pull/2230
[2689] https://github.com/STEllAR-GROUP/hpx/pull/2229
[2690] https://github.com/STEllAR-GROUP/hpx/pull/2228

- PR #2227[2691] - Support for HPXCL's opencl::event
- PR #2226[2692] - Preparing for release of V0.9.99
- PR #2225[2693] - fix issue when compiling components with hpxcxx
- PR #2224[2694] - Compute alloc fix
- PR #2223[2695] - Simplify promise
- PR #2222[2696] - Replace last uses of boost::function by util::function_nonser
- PR #2221[2697] - Fix config tests
- PR #2220[2698] - Fixing gcc 4.6 compilation issues
- PR #2219[2699] - nullptr support for `[unique_]function`
- PR #2218[2700] - Introducing clang tidy
- PR #2216[2701] - Replace NULL with nullptr
- Issue #2214[2702] - Let inspect flag use of NULL, suggest nullptr instead
- PR #2213[2703] - Require support for nullptr
- PR #2212[2704] - Properly find jemalloc through pkg-config
- PR #2211[2705] - Disable a couple of warnings reported by Intel on Windows
- PR #2210[2706] - Fixed host::block_allocator::bulk_construct
- PR #2209[2707] - Started to clean up new sort algorithms, made things compile for sort_by_key
- PR #2208[2708] - A couple of fixes that were exposed by a new sort algorithm
- PR #2207[2709] - Adding missing includes in /hpx/include/serialization.hpp
- PR #2206[2710] - Call package_action::get_future before package_action::apply
- PR #2205[2711] - The indirect_packaged_task::operator() needs to be run on a HPX thread
- PR #2204[2712] - Variadic executor parameters
- PR #2203[2713] - Delay-initialize members of partitoned iterator

2691 https://github.com/STEllAR-GROUP/hpx/pull/2227
2692 https://github.com/STEllAR-GROUP/hpx/pull/2226
2693 https://github.com/STEllAR-GROUP/hpx/pull/2225
2694 https://github.com/STEllAR-GROUP/hpx/pull/2224
2695 https://github.com/STEllAR-GROUP/hpx/pull/2223
2696 https://github.com/STEllAR-GROUP/hpx/pull/2222
2697 https://github.com/STEllAR-GROUP/hpx/pull/2221
2698 https://github.com/STEllAR-GROUP/hpx/pull/2220
2699 https://github.com/STEllAR-GROUP/hpx/pull/2219
2700 https://github.com/STEllAR-GROUP/hpx/pull/2218
2701 https://github.com/STEllAR-GROUP/hpx/pull/2216
2702 https://github.com/STEllAR-GROUP/hpx/issues/2214
2703 https://github.com/STEllAR-GROUP/hpx/pull/2213
2704 https://github.com/STEllAR-GROUP/hpx/pull/2212
2705 https://github.com/STEllAR-GROUP/hpx/pull/2211
2706 https://github.com/STEllAR-GROUP/hpx/pull/2210
2707 https://github.com/STEllAR-GROUP/hpx/pull/2209
2708 https://github.com/STEllAR-GROUP/hpx/pull/2208
2709 https://github.com/STEllAR-GROUP/hpx/pull/2207
2710 https://github.com/STEllAR-GROUP/hpx/pull/2206
2711 https://github.com/STEllAR-GROUP/hpx/pull/2205
2712 https://github.com/STEllAR-GROUP/hpx/pull/2204
2713 https://github.com/STEllAR-GROUP/hpx/pull/2203

- PR #2202[2714] - Added segmented fill for hpx::vector
- Issue #2201[2715] - Null Thread id encountered on partitioned_vector
- PR #2200[2716] - Fix hangs
- PR #2199[2717] - Deprecating hpx/traits.hpp
- PR #2198[2718] - Making explicit inclusion of external libraries into build
- PR #2197[2719] - Fix typo in QT CMakeLists
- PR #2196[2720] - Fixing a gcc warning about attributes being ignored
- PR #2194[2721] - Fixing partitioned_vector_spmd_foreach example
- Issue #2193[2722] - partitioned_vector_spmd_foreach seg faults
- PR #2192[2723] - Support Boost.Thread v4
- PR #2191[2724] - HPX.Compute prototype
- PR #2190[2725] - Spawning operation on new thread if remaining stack space becomes too small
- PR #2189[2726] - Adding callback taking index and future to when_each
- PR #2188[2727] - Adding new example demonstrating receive_buffer
- PR #2187[2728] - Mask 128-bit ints if CUDA is being used
- PR #2186[2729] - Make startup & shutdown functions unique_function
- PR #2185[2730] - Fixing logging output not to cause hang on shutdown
- PR #2184[2731] - Allowing component clients as action return types
- Issue #2183[2732] - Enabling logging output causes hang on shutdown
- Issue #2182[2733] - 1d_stencil seg fault
- Issue #2181[2734] - Setting small stack size does not change default
- PR #2180[2735] - Changing default bind mode to balanced
- PR #2179[2736] - adding prefetching_iterator and related tests used for prefetching containers within lambda functions

[2714] https://github.com/STEllAR-GROUP/hpx/pull/2202
[2715] https://github.com/STEllAR-GROUP/hpx/issues/2201
[2716] https://github.com/STEllAR-GROUP/hpx/pull/2200
[2717] https://github.com/STEllAR-GROUP/hpx/pull/2199
[2718] https://github.com/STEllAR-GROUP/hpx/pull/2198
[2719] https://github.com/STEllAR-GROUP/hpx/pull/2197
[2720] https://github.com/STEllAR-GROUP/hpx/pull/2196
[2721] https://github.com/STEllAR-GROUP/hpx/pull/2194
[2722] https://github.com/STEllAR-GROUP/hpx/issues/2193
[2723] https://github.com/STEllAR-GROUP/hpx/pull/2192
[2724] https://github.com/STEllAR-GROUP/hpx/pull/2191
[2725] https://github.com/STEllAR-GROUP/hpx/pull/2190
[2726] https://github.com/STEllAR-GROUP/hpx/pull/2189
[2727] https://github.com/STEllAR-GROUP/hpx/pull/2188
[2728] https://github.com/STEllAR-GROUP/hpx/pull/2187
[2729] https://github.com/STEllAR-GROUP/hpx/pull/2186
[2730] https://github.com/STEllAR-GROUP/hpx/pull/2185
[2731] https://github.com/STEllAR-GROUP/hpx/pull/2184
[2732] https://github.com/STEllAR-GROUP/hpx/issues/2183
[2733] https://github.com/STEllAR-GROUP/hpx/issues/2182
[2734] https://github.com/STEllAR-GROUP/hpx/issues/2181
[2735] https://github.com/STEllAR-GROUP/hpx/pull/2180
[2736] https://github.com/STEllAR-GROUP/hpx/pull/2179

- PR #2177[2737] - Fixing 2176
- Issue #2176[2738] - Launch process test fails on OSX
- PR #2175[2739] - Fix unbalanced config/warnings includes, add some new ones
- PR #2174[2740] - Fix test categorization : regression not unit
- Issue #2172[2741] - Different performance results
- Issue #2171[2742] - "negative entry in reference count table" running octotiger on 32 nodes on queenbee
- Issue #2170[2743] - Error while compiling on Mac + boost 1.60
- PR #2168[2744] - Fixing problems with is_bitwise_serializable
- Issue #2167[2745] - startup & shutdown function should accept unique_function
- Issue #2166[2746] - Simple receive_buffer example
- PR #2165[2747] - Fix wait all
- PR #2164[2748] - Fix wait all
- PR #2163[2749] - Fix some typos in config tests
- PR #2162[2750] - Improve #includes
- PR #2160[2751] - Add inspect check for missing #include <list>
- PR #2159[2752] - Add missing finalize call to stop test hanging
- PR #2158[2753] - Algo fixes
- PR #2157[2754] - Stack check
- Issue #2156[2755] - OSX reports stack space incorrectly (generic context coroutines)
- Issue #2155[2756] - Race condition suspected in runtime
- PR #2154[2757] - Replace boost::detail::atomic_count with the new util::atomic_count
- PR #2153[2758] - Fix stack overflow on OSX
- PR #2152[2759] - Define is_bitwise_serializable as is_trivially_copyable when available

2737 https://github.com/STEllAR-GROUP/hpx/pull/2177
2738 https://github.com/STEllAR-GROUP/hpx/issues/2176
2739 https://github.com/STEllAR-GROUP/hpx/pull/2175
2740 https://github.com/STEllAR-GROUP/hpx/pull/2174
2741 https://github.com/STEllAR-GROUP/hpx/issues/2172
2742 https://github.com/STEllAR-GROUP/hpx/issues/2171
2743 https://github.com/STEllAR-GROUP/hpx/issues/2170
2744 https://github.com/STEllAR-GROUP/hpx/pull/2168
2745 https://github.com/STEllAR-GROUP/hpx/issues/2167
2746 https://github.com/STEllAR-GROUP/hpx/issues/2166
2747 https://github.com/STEllAR-GROUP/hpx/pull/2165
2748 https://github.com/STEllAR-GROUP/hpx/pull/2164
2749 https://github.com/STEllAR-GROUP/hpx/pull/2163
2750 https://github.com/STEllAR-GROUP/hpx/pull/2162
2751 https://github.com/STEllAR-GROUP/hpx/pull/2160
2752 https://github.com/STEllAR-GROUP/hpx/pull/2159
2753 https://github.com/STEllAR-GROUP/hpx/pull/2158
2754 https://github.com/STEllAR-GROUP/hpx/pull/2157
2755 https://github.com/STEllAR-GROUP/hpx/issues/2156
2756 https://github.com/STEllAR-GROUP/hpx/issues/2155
2757 https://github.com/STEllAR-GROUP/hpx/pull/2154
2758 https://github.com/STEllAR-GROUP/hpx/pull/2153
2759 https://github.com/STEllAR-GROUP/hpx/pull/2152

- PR #2151[2760] - Adding missing <cstring> for std::mem* functions
- Issue #2150[2761] - Unable to use component clients as action return types
- PR #2149[2762] - std::memmove copies bytes, use bytes*sizeof(type) when copying larger types
- PR #2146[2763] - Adding customization point for parallel copy/move
- PR #2145[2764] - Applying changes to address warnings issued by latest version of PVS Studio
- Issue #2148[2765] - hpx::parallel::copy is broken after trivially copyable changes
- PR #2144[2766] - Some minor tweaks to compute prototype
- PR #2143[2767] - Added Boost version support information over OSX platform
- PR #2142[2768] - Fixing memory leak in example
- PR #2141[2769] - Add missing specializations in execution policies
- PR #2139[2770] - This PR fixes a few problems reported by Clang's Undefined Behavior sanitizer
- PR #2138[2771] - Revert "Adding fedora docs"
- PR #2136[2772] - Removed double semicolon
- PR #2135[2773] - Add deprecated #include check for hpx_fwd.hpp
- PR #2134[2774] - Resolved memory leak in stencil_8
- PR #2133[2775] - Replace uses of boost pointer containers
- PR #2132[2776] - Removing unused typedef
- PR #2131[2777] - Add several include checks for std facilities
- PR #2130[2778] - Fixing parcel compression, adding test
- PR #2129[2779] - Fix invalid attribute warnings
- Issue #2128[2780] - hpx::init seems to segfault
- PR #2127[2781] - Making executor_traits N-nary
- PR #2126[2782] - GCC 4.6 fails to deduce the correct type in lambda

[2760] https://github.com/STEllAR-GROUP/hpx/pull/2151
[2761] https://github.com/STEllAR-GROUP/hpx/issues/2150
[2762] https://github.com/STEllAR-GROUP/hpx/pull/2149
[2763] https://github.com/STEllAR-GROUP/hpx/pull/2146
[2764] https://github.com/STEllAR-GROUP/hpx/pull/2145
[2765] https://github.com/STEllAR-GROUP/hpx/issues/2148
[2766] https://github.com/STEllAR-GROUP/hpx/pull/2144
[2767] https://github.com/STEllAR-GROUP/hpx/pull/2143
[2768] https://github.com/STEllAR-GROUP/hpx/pull/2142
[2769] https://github.com/STEllAR-GROUP/hpx/pull/2141
[2770] https://github.com/STEllAR-GROUP/hpx/pull/2139
[2771] https://github.com/STEllAR-GROUP/hpx/pull/2138
[2772] https://github.com/STEllAR-GROUP/hpx/pull/2136
[2773] https://github.com/STEllAR-GROUP/hpx/pull/2135
[2774] https://github.com/STEllAR-GROUP/hpx/pull/2134
[2775] https://github.com/STEllAR-GROUP/hpx/pull/2133
[2776] https://github.com/STEllAR-GROUP/hpx/pull/2132
[2777] https://github.com/STEllAR-GROUP/hpx/pull/2131
[2778] https://github.com/STEllAR-GROUP/hpx/pull/2130
[2779] https://github.com/STEllAR-GROUP/hpx/pull/2129
[2780] https://github.com/STEllAR-GROUP/hpx/issues/2128
[2781] https://github.com/STEllAR-GROUP/hpx/pull/2127
[2782] https://github.com/STEllAR-GROUP/hpx/pull/2126

- PR #2125[2783] - Making parcel coalescing test actually test something
- Issue #2124[2784] - Make a testcase for parcel compression
- Issue #2123[2785] - hpx/hpx/runtime/applier_fwd.hpp - Multiple defined types
- Issue #2122[2786] - Exception in primary_namespace::resolve_free_list
- Issue #2121[2787] - Possible memory leak in 1d_stencil_8
- PR #2120[2788] - Fixing 2119
- Issue #2119[2789] - reduce_by_key compilation problems
- Issue #2118[2790] - Premature unwrapping of boost::ref'ed arguments
- PR #2117[2791] - Added missing initializer on last constructor for thread_description
- PR #2116[2792] - Use a lightweight bind implementation when no placeholders are given
- PR #2115[2793] - Replace boost::shared_ptr with std::shared_ptr
- PR #2114[2794] - Adding hook functions for executor_parameter_traits supporting timers
- Issue #2113[2795] - Compilation error with gcc version 4.9.3 (MacPorts gcc49 4.9.3_0)
- PR #2112[2796] - Replace uses of safe_bool with explicit operator bool
- Issue #2111[2797] - Compilation error on QT example
- Issue #2110[2798] - Compilation error when passing non-future argument to unwrapped continuation in dataflow
- Issue #2109[2799] - Warning while compiling hpx
- Issue #2109[2800] - Stack trace of last bug causing issues with octotiger
- Issue #2108[2801] - Stack trace of last bug causing issues with octotiger
- PR #2107[2802] - Making sure that a missing parcel_coalescing module does not cause startup exceptions
- PR #2106[2803] - Stop using hpx_fwd.hpp
- Issue #2105[2804] - coalescing plugin handler is not optional any more
- Issue #2104[2805] - Make executor_traits N-nary

2783 https://github.com/STEllAR-GROUP/hpx/pull/2125
2784 https://github.com/STEllAR-GROUP/hpx/issues/2124
2785 https://github.com/STEllAR-GROUP/hpx/issues/2123
2786 https://github.com/STEllAR-GROUP/hpx/issues/2122
2787 https://github.com/STEllAR-GROUP/hpx/issues/2121
2788 https://github.com/STEllAR-GROUP/hpx/pull/2120
2789 https://github.com/STEllAR-GROUP/hpx/issues/2119
2790 https://github.com/STEllAR-GROUP/hpx/issues/2118
2791 https://github.com/STEllAR-GROUP/hpx/pull/2117
2792 https://github.com/STEllAR-GROUP/hpx/pull/2116
2793 https://github.com/STEllAR-GROUP/hpx/pull/2115
2794 https://github.com/STEllAR-GROUP/hpx/pull/2114
2795 https://github.com/STEllAR-GROUP/hpx/issues/2113
2796 https://github.com/STEllAR-GROUP/hpx/pull/2112
2797 https://github.com/STEllAR-GROUP/hpx/issues/2111
2798 https://github.com/STEllAR-GROUP/hpx/issues/2110
2799 https://github.com/STEllAR-GROUP/hpx/issues/2109
2800 https://github.com/STEllAR-GROUP/hpx/issues/2109
2801 https://github.com/STEllAR-GROUP/hpx/issues/2108
2802 https://github.com/STEllAR-GROUP/hpx/pull/2107
2803 https://github.com/STEllAR-GROUP/hpx/pull/2106
2804 https://github.com/STEllAR-GROUP/hpx/issues/2105
2805 https://github.com/STEllAR-GROUP/hpx/issues/2104

- Issue #2103[2806] - Build error with octotiger and hpx commit e657426d
- PR #2102[2807] - Combining thread data storage
- PR #2101[2808] - Added repartition version of 1d stencil that uses any performance counter
- PR #2100[2809] - Drop obsolete TR1 result_of protocol
- PR #2099[2810] - Replace uses of boost::bind with util::bind
- PR #2098[2811] - Deprecated inspect checks
- PR #2097[2812] - Reduce by key, extends #1141
- PR #2096[2813] - Moving local cache from external to hpx/util
- PR #2095[2814] - Bump minimum required Boost to 1.50.0
- PR #2094[2815] - Add include checks for several Boost utilities
- Issue #2093[2816] - /.../local_cache.hpp(89): error #303: explicit type is missing ("int" assumed)
- PR #2091[2817] - Fix for Raspberry pi build
- PR #2090[2818] - Fix storage size for util::function<>
- PR #2089[2819] - Fix #2088
- Issue #2088[2820] - More verbose output from cmake configuration
- PR #2087[2821] - Making sure init_globally always executes hpx_main
- Issue #2086[2822] - Race condition with recent HPX
- PR #2085[2823] - Adding #include checker
- PR #2084[2824] - Replace boost lock types with standard library ones
- PR #2083[2825] - Simplify packaged task
- PR #2082[2826] - Updating APEX version for testing
- PR #2081[2827] - Cleanup exception headers
- PR #2080[2828] - Make call_once variadic

2806 https://github.com/STEllAR-GROUP/hpx/issues/2103
2807 https://github.com/STEllAR-GROUP/hpx/pull/2102
2808 https://github.com/STEllAR-GROUP/hpx/pull/2101
2809 https://github.com/STEllAR-GROUP/hpx/pull/2100
2810 https://github.com/STEllAR-GROUP/hpx/pull/2099
2811 https://github.com/STEllAR-GROUP/hpx/pull/2098
2812 https://github.com/STEllAR-GROUP/hpx/pull/2097
2813 https://github.com/STEllAR-GROUP/hpx/pull/2096
2814 https://github.com/STEllAR-GROUP/hpx/pull/2095
2815 https://github.com/STEllAR-GROUP/hpx/pull/2094
2816 https://github.com/STEllAR-GROUP/hpx/issues/2093
2817 https://github.com/STEllAR-GROUP/hpx/pull/2091
2818 https://github.com/STEllAR-GROUP/hpx/pull/2090
2819 https://github.com/STEllAR-GROUP/hpx/pull/2089
2820 https://github.com/STEllAR-GROUP/hpx/issues/2088
2821 https://github.com/STEllAR-GROUP/hpx/pull/2087
2822 https://github.com/STEllAR-GROUP/hpx/issues/2086
2823 https://github.com/STEllAR-GROUP/hpx/pull/2085
2824 https://github.com/STEllAR-GROUP/hpx/pull/2084
2825 https://github.com/STEllAR-GROUP/hpx/pull/2083
2826 https://github.com/STEllAR-GROUP/hpx/pull/2082
2827 https://github.com/STEllAR-GROUP/hpx/pull/2081
2828 https://github.com/STEllAR-GROUP/hpx/pull/2080

- Issue #2079[2829] - With GNU C++, line 85 of hpx/config/version.hpp causes link failure when linking application
- Issue #2078[2830] - Simple test fails with _GLIBCXX_DEBUG defined
- PR #2077[2831] - Instantiate board in nqueen client
- PR #2076[2832] - Moving coalescing registration to TUs
- PR #2075[2833] - Fixed some documentation typos
- PR #2074[2834] - Adding flush-mode to message handler flush
- PR #2073[2835] - Fixing performance regression introduced lately
- PR #2072[2836] - Refactor local::condition_variable
- PR #2071[2837] - Timer based on boost::asio::deadline_timer
- PR #2070[2838] - Refactor tuple based functionality
- PR #2069[2839] - Fixed typos
- Issue #2068[2840] - Seg fault with octotiger
- PR #2067[2841] - Algorithm cleanup
- PR #2066[2842] - Split credit fixes
- PR #2065[2843] - Rename HPX_MOVABLE_BUT_NOT_COPYABLE to HPX_MOVABLE_ONLY
- PR #2064[2844] - Fixed some typos in docs
- PR #2063[2845] - Adding example demonstrating template components
- Issue #2062[2846] - Support component templates
- PR #2061[2847] - Replace some uses of lexical_cast<string> with C++11 std::to_string
- PR #2060[2848] - Replace uses of boost::noncopyable with HPX_NON_COPYABLE
- PR #2059[2849] - Adding missing for_loop algorithms
- PR #2058[2850] - Move several definitions to more appropriate headers
- PR #2057[2851] - Simplify assert_owns_lock and ignore_while_checking

2829 https://github.com/STEllAR-GROUP/hpx/issues/2079
2830 https://github.com/STEllAR-GROUP/hpx/issues/2078
2831 https://github.com/STEllAR-GROUP/hpx/pull/2077
2832 https://github.com/STEllAR-GROUP/hpx/pull/2076
2833 https://github.com/STEllAR-GROUP/hpx/pull/2075
2834 https://github.com/STEllAR-GROUP/hpx/pull/2074
2835 https://github.com/STEllAR-GROUP/hpx/pull/2073
2836 https://github.com/STEllAR-GROUP/hpx/pull/2072
2837 https://github.com/STEllAR-GROUP/hpx/pull/2071
2838 https://github.com/STEllAR-GROUP/hpx/pull/2070
2839 https://github.com/STEllAR-GROUP/hpx/pull/2069
2840 https://github.com/STEllAR-GROUP/hpx/issues/2068
2841 https://github.com/STEllAR-GROUP/hpx/pull/2067
2842 https://github.com/STEllAR-GROUP/hpx/pull/2066
2843 https://github.com/STEllAR-GROUP/hpx/pull/2065
2844 https://github.com/STEllAR-GROUP/hpx/pull/2064
2845 https://github.com/STEllAR-GROUP/hpx/pull/2063
2846 https://github.com/STEllAR-GROUP/hpx/issues/2062
2847 https://github.com/STEllAR-GROUP/hpx/pull/2061
2848 https://github.com/STEllAR-GROUP/hpx/pull/2060
2849 https://github.com/STEllAR-GROUP/hpx/pull/2059
2850 https://github.com/STEllAR-GROUP/hpx/pull/2058
2851 https://github.com/STEllAR-GROUP/hpx/pull/2057

- PR #2056[2852] - Replacing std::result_of with util::result_of
- PR #2055[2853] - Fix process launching/connecting back
- PR #2054[2854] - Add a forwarding coroutine header
- PR #2053[2855] - Replace uses of boost::unordered_map with std::unordered_map
- PR #2052[2856] - Rewrite tuple unwrap
- PR #2050[2857] - Replace uses of BOOST_SCOPED_ENUM with C++11 scoped enums
- PR #2049[2858] - Attempt to narrow down split_credit problem
- PR #2048[2859] - Fixing gcc startup hangs
- PR #2047[2860] - Fixing when_xxx and wait_xxx for MSVC12
- PR #2046[2861] - adding persistent_auto_chunk_size and related tests for for_each
- PR #2045[2862] - Fixing HPX_HAVE_THREAD_BACKTRACE_DEPTH build time configuration
- PR #2044[2863] - Adding missing service executor types
- PR #2043[2864] - Removing ambiguous definitions for is_future_range and future_range_traits
- PR #2042[2865] - Clarify that HPX builds can use (much) more than 2GB per process
- PR #2041[2866] - Changing future_iterator_traits to support pointers
- Issue #2040[2867] - Improve documentation memory usage warning?
- PR #2039[2868] - Coroutine cleanup
- PR #2038[2869] - Fix cmake policy CMP0042 warning MACOSX_RPATH
- PR #2037[2870] - Avoid redundant specialization of [**unique_**]function_nonser
- PR #2036[2871] - nvcc dies with an internal error upon pushing/popping warnings inside templates
- Issue #2035[2872] - Use a less restrictive iterator definition in hpx::lcos::detail::future_iterator_traits
- PR #2034[2873] - Fixing compilation error with thread queue wait time performance counter
- Issue #2033[2874] - Compilation error when compiling with thread queue waittime performance counter

2852 https://github.com/STEllAR-GROUP/hpx/pull/2056
2853 https://github.com/STEllAR-GROUP/hpx/pull/2055
2854 https://github.com/STEllAR-GROUP/hpx/pull/2054
2855 https://github.com/STEllAR-GROUP/hpx/pull/2053
2856 https://github.com/STEllAR-GROUP/hpx/pull/2052
2857 https://github.com/STEllAR-GROUP/hpx/pull/2050
2858 https://github.com/STEllAR-GROUP/hpx/pull/2049
2859 https://github.com/STEllAR-GROUP/hpx/pull/2048
2860 https://github.com/STEllAR-GROUP/hpx/pull/2047
2861 https://github.com/STEllAR-GROUP/hpx/pull/2046
2862 https://github.com/STEllAR-GROUP/hpx/pull/2045
2863 https://github.com/STEllAR-GROUP/hpx/pull/2044
2864 https://github.com/STEllAR-GROUP/hpx/pull/2043
2865 https://github.com/STEllAR-GROUP/hpx/pull/2042
2866 https://github.com/STEllAR-GROUP/hpx/pull/2041
2867 https://github.com/STEllAR-GROUP/hpx/issues/2040
2868 https://github.com/STEllAR-GROUP/hpx/pull/2039
2869 https://github.com/STEllAR-GROUP/hpx/pull/2038
2870 https://github.com/STEllAR-GROUP/hpx/pull/2037
2871 https://github.com/STEllAR-GROUP/hpx/pull/2036
2872 https://github.com/STEllAR-GROUP/hpx/issues/2035
2873 https://github.com/STEllAR-GROUP/hpx/pull/2034
2874 https://github.com/STEllAR-GROUP/hpx/issues/2033

- Issue #2032[2875] - Ambiguous template instantiation for is_future_range and future_range_traits.
- PR #2031[2876] - Don't restart timer on every incoming parcel
- PR #2030[2877] - Unify handling of execution policies in parallel algorithms
- PR #2029[2878] - Make pkg-config .pc files use .dylib on OSX
- PR #2028[2879] - Adding process component
- PR #2027[2880] - Making check for compiler compatibility independent on compiler path
- PR #2025[2881] - Fixing inspect tool
- PR #2024[2882] - Intel13 removal
- PR #2023[2883] - Fix errors related to older boost versions and parameter pack expansions in lambdas
- Issue #2022[2884] - gmake fail: "No rule to make target /usr/lib46/libboost_context-mt.so"
- PR #2021[2885] - Added Sudoku example
- Issue #2020[2886] - Make errors related to init_globally.cpp example while building HPX out of the box
- PR #2019[2887] - Fixed some compilation and cmake errors encountered in nqueen example
- PR #2018[2888] - For loop algorithms
- PR #2017[2889] - Non-recursive at_index implementation
- Issue #2016[2890] - Add index-based for-loops
- Issue #2015[2891] - Change default bind-mode to balanced
- PR #2014[2892] - Fixed dataflow if invoked action returns a future
- PR #2013[2893] - Fixing compilation issues with external example
- PR #2012[2894] - Added Sierpinski Triangle example
- Issue #2011[2895] - Compilation error while running sample hello_world_component code
- PR #2010[2896] - Segmented move implemented for hpx::vector
- Issue #2009[2897] - pkg-config order incorrect on 14.04 / GCC 4.8

[2875] https://github.com/STEllAR-GROUP/hpx/issues/2032
[2876] https://github.com/STEllAR-GROUP/hpx/pull/2031
[2877] https://github.com/STEllAR-GROUP/hpx/pull/2030
[2878] https://github.com/STEllAR-GROUP/hpx/pull/2029
[2879] https://github.com/STEllAR-GROUP/hpx/pull/2028
[2880] https://github.com/STEllAR-GROUP/hpx/pull/2027
[2881] https://github.com/STEllAR-GROUP/hpx/pull/2025
[2882] https://github.com/STEllAR-GROUP/hpx/pull/2024
[2883] https://github.com/STEllAR-GROUP/hpx/pull/2023
[2884] https://github.com/STEllAR-GROUP/hpx/issues/2022
[2885] https://github.com/STEllAR-GROUP/hpx/pull/2021
[2886] https://github.com/STEllAR-GROUP/hpx/issues/2020
[2887] https://github.com/STEllAR-GROUP/hpx/pull/2019
[2888] https://github.com/STEllAR-GROUP/hpx/pull/2018
[2889] https://github.com/STEllAR-GROUP/hpx/pull/2017
[2890] https://github.com/STEllAR-GROUP/hpx/issues/2016
[2891] https://github.com/STEllAR-GROUP/hpx/issues/2015
[2892] https://github.com/STEllAR-GROUP/hpx/pull/2014
[2893] https://github.com/STEllAR-GROUP/hpx/pull/2013
[2894] https://github.com/STEllAR-GROUP/hpx/pull/2012
[2895] https://github.com/STEllAR-GROUP/hpx/issues/2011
[2896] https://github.com/STEllAR-GROUP/hpx/pull/2010
[2897] https://github.com/STEllAR-GROUP/hpx/issues/2009

- Issue #2008[2898] - Compilation error in dataflow of action returning a future
- PR #2007[2899] - Adding new performance counter exposing overall scheduler time
- PR #2006[2900] - Function includes
- PR #2005[2901] - Adding an example demonstrating how to initialize HPX from a global object
- PR #2004[2902] - Fixing 2000
- PR #2003[2903] - Adding generation parameter to gather to enable using it more than once
- PR #2002[2904] - Turn on position independent code to solve link problem with hpx_init
- Issue #2001[2905] - Gathering more than once segfaults
- Issue #2000[2906] - Undefined reference to hpx::assertion_failed
- Issue #1999[2907] - Seg fault in hpx::lcos::base_lco_with_value<*>::set_value_nonvirt() when running octo-tiger
- PR #1998[2908] - Detect unknown command line options
- PR #1997[2909] - Extending thread description
- PR #1996[2910] - Adding natvis files to solution (MSVC only)
- Issue #1995[2911] - Command line handling does not produce error
- PR #1994[2912] - Possible missing include in test_utils.hpp
- PR #1993[2913] - Add missing LANGUAGES tag to a hpx_add_compile_flag_if_available() call in CMakeLists.txt
- PR #1992[2914] - Fixing shared_executor_test
- PR #1991[2915] - Making sure the winsock library is properly initialized
- PR #1990[2916] - Fixing bind_test placeholder ambiguity coming from boost-1.60
- PR #1989[2917] - Performance tuning
- PR #1987[2918] - Make configurable size of internal storage in util::function
- PR #1986[2919] - AGAS Refactoring+1753 Cache mods
- PR #1985[2920] - Adding missing task_block::run() overload taking an executor

[2898] https://github.com/STEllAR-GROUP/hpx/issues/2008
[2899] https://github.com/STEllAR-GROUP/hpx/pull/2007
[2900] https://github.com/STEllAR-GROUP/hpx/pull/2006
[2901] https://github.com/STEllAR-GROUP/hpx/pull/2005
[2902] https://github.com/STEllAR-GROUP/hpx/pull/2004
[2903] https://github.com/STEllAR-GROUP/hpx/pull/2003
[2904] https://github.com/STEllAR-GROUP/hpx/pull/2002
[2905] https://github.com/STEllAR-GROUP/hpx/issues/2001
[2906] https://github.com/STEllAR-GROUP/hpx/issues/2000
[2907] https://github.com/STEllAR-GROUP/hpx/issues/1999
[2908] https://github.com/STEllAR-GROUP/hpx/pull/1998
[2909] https://github.com/STEllAR-GROUP/hpx/pull/1997
[2910] https://github.com/STEllAR-GROUP/hpx/pull/1996
[2911] https://github.com/STEllAR-GROUP/hpx/issues/1995
[2912] https://github.com/STEllAR-GROUP/hpx/pull/1994
[2913] https://github.com/STEllAR-GROUP/hpx/pull/1993
[2914] https://github.com/STEllAR-GROUP/hpx/pull/1992
[2915] https://github.com/STEllAR-GROUP/hpx/pull/1991
[2916] https://github.com/STEllAR-GROUP/hpx/pull/1990
[2917] https://github.com/STEllAR-GROUP/hpx/pull/1989
[2918] https://github.com/STEllAR-GROUP/hpx/pull/1987
[2919] https://github.com/STEllAR-GROUP/hpx/pull/1986
[2920] https://github.com/STEllAR-GROUP/hpx/pull/1985

- PR #1984[2921] - Adding an optimized LRU Cache implementation (for AGAS)
- PR #1983[2922] - Avoid invoking migration table look up for all objects
- PR #1981[2923] - Replacing uintptr_t (which is not defined everywhere) with std::size_t
- PR #1980[2924] - Optimizing LCO continuations
- PR #1979[2925] - Fixing Cori
- PR #1978[2926] - Fix test check that got broken in hasty fix to memory overflow
- PR #1977[2927] - Refactor action traits
- PR #1976[2928] - Fixes typo in README.rst
- PR #1975[2929] - Reduce size of benchmark timing arrays to fix test failures
- PR #1974[2930] - Add action to update data owned by the partitioned_vector component
- PR #1972[2931] - Adding partitioned_vector SPMD example
- PR #1971[2932] - Fixing 1965
- PR #1970[2933] - Papi fixes
- PR #1969[2934] - Fixing continuation recursions to not depend on fixed amount of recursions
- PR #1968[2935] - More segmented algorithms
- Issue #1967[2936] - Simplify component implementations
- PR #1966[2937] - Migrate components
- Issue #1964[2938] - fatal error: 'boost/lockfree/detail/branch_hints.hpp' file not found
- Issue #1962[2939] - parallel:copy_if has race condition when used on in place arrays
- PR #1963[2940] - Fixing Static Parcelport initialization
- PR #1961[2941] - Fix function target
- Issue #1960[2942] - Papi counters don't reset
- PR #1959[2943] - Fixing 1958

2921 https://github.com/STEllAR-GROUP/hpx/pull/1984
2922 https://github.com/STEllAR-GROUP/hpx/pull/1983
2923 https://github.com/STEllAR-GROUP/hpx/pull/1981
2924 https://github.com/STEllAR-GROUP/hpx/pull/1980
2925 https://github.com/STEllAR-GROUP/hpx/pull/1979
2926 https://github.com/STEllAR-GROUP/hpx/pull/1978
2927 https://github.com/STEllAR-GROUP/hpx/pull/1977
2928 https://github.com/STEllAR-GROUP/hpx/pull/1976
2929 https://github.com/STEllAR-GROUP/hpx/pull/1975
2930 https://github.com/STEllAR-GROUP/hpx/pull/1974
2931 https://github.com/STEllAR-GROUP/hpx/pull/1972
2932 https://github.com/STEllAR-GROUP/hpx/pull/1971
2933 https://github.com/STEllAR-GROUP/hpx/pull/1970
2934 https://github.com/STEllAR-GROUP/hpx/pull/1969
2935 https://github.com/STEllAR-GROUP/hpx/pull/1968
2936 https://github.com/STEllAR-GROUP/hpx/issues/1967
2937 https://github.com/STEllAR-GROUP/hpx/pull/1966
2938 https://github.com/STEllAR-GROUP/hpx/issues/1964
2939 https://github.com/STEllAR-GROUP/hpx/issues/1962
2940 https://github.com/STEllAR-GROUP/hpx/pull/1963
2941 https://github.com/STEllAR-GROUP/hpx/pull/1961
2942 https://github.com/STEllAR-GROUP/hpx/issues/1960
2943 https://github.com/STEllAR-GROUP/hpx/pull/1959

- Issue #1958[2944] - inclusive_scan gives incorrect results with non-commutative operator
- PR #1957[2945] - Fixing #1950
- PR #1956[2946] - Sort by key example
- PR #1955[2947] - Adding regression test for #1946: Hang in wait_all() in distributed run
- Issue #1954[2948] - HPX releases should not use -Werror
- PR #1953[2949] - Adding performance analysis for AGAS cache
- PR #1952[2950] - Adapting test for explicit variadics to fail for gcc 4.6
- PR #1951[2951] - Fixing memory leak
- Issue #1950[2952] - Simplify external builds
- PR #1949[2953] - Fixing yet another lock that is being held during suspension
- PR #1948[2954] - Fixed container algorithms for Intel
- PR #1947[2955] - Adding workaround for tagged_tuple
- Issue #1946[2956] - Hang in wait_all() in distributed run
- PR #1945[2957] - Fixed container algorithm tests
- Issue #1944[2958] - assertion 'p.destination_locality() == hpx::get_locality()' failed
- PR #1943[2959] - Fix a couple of compile errors with clang
- PR #1942[2960] - Making parcel coalescing functional
- Issue #1941[2961] - Re-enable parcel coalescing
- PR #1940[2962] - Touching up make_future
- PR #1939[2963] - Fixing problems in over-subscription management in the resource manager
- PR #1938[2964] - Removing use of unified Boost.Thread header
- PR #1937[2965] - Cleaning up the use of Boost.Accumulator headers
- PR #1936[2966] - Making sure interval timer is started for aggregating performance counters

[2944] https://github.com/STEllAR-GROUP/hpx/issues/1958
[2945] https://github.com/STEllAR-GROUP/hpx/pull/1957
[2946] https://github.com/STEllAR-GROUP/hpx/pull/1956
[2947] https://github.com/STEllAR-GROUP/hpx/pull/1955
[2948] https://github.com/STEllAR-GROUP/hpx/issues/1954
[2949] https://github.com/STEllAR-GROUP/hpx/pull/1953
[2950] https://github.com/STEllAR-GROUP/hpx/pull/1952
[2951] https://github.com/STEllAR-GROUP/hpx/pull/1951
[2952] https://github.com/STEllAR-GROUP/hpx/issues/1950
[2953] https://github.com/STEllAR-GROUP/hpx/pull/1949
[2954] https://github.com/STEllAR-GROUP/hpx/pull/1948
[2955] https://github.com/STEllAR-GROUP/hpx/pull/1947
[2956] https://github.com/STEllAR-GROUP/hpx/issues/1946
[2957] https://github.com/STEllAR-GROUP/hpx/pull/1945
[2958] https://github.com/STEllAR-GROUP/hpx/issues/1944
[2959] https://github.com/STEllAR-GROUP/hpx/pull/1943
[2960] https://github.com/STEllAR-GROUP/hpx/pull/1942
[2961] https://github.com/STEllAR-GROUP/hpx/issues/1941
[2962] https://github.com/STEllAR-GROUP/hpx/pull/1940
[2963] https://github.com/STEllAR-GROUP/hpx/pull/1939
[2964] https://github.com/STEllAR-GROUP/hpx/pull/1938
[2965] https://github.com/STEllAR-GROUP/hpx/pull/1937
[2966] https://github.com/STEllAR-GROUP/hpx/pull/1936

- PR #1935[2967] - Tagged results
- PR #1934[2968] - Fix remote async with deferred launch policy
- Issue #1933[2969] - Floating point exception in `statistics_counter<boost::accumulators::tag::mean>::get_c`
- PR #1932[2970] - Removing superfluous includes of boost/lockfree/detail/branch_hints.hpp
- PR #1931[2971] - fix compilation with clang 3.8.0
- Issue #1930[2972] - Missing online documentation for HPX 0.9.11
- PR #1929[2973] - LWG2485: get() should be overloaded for const tuple&&
- PR #1928[2974] - Revert "Using ninja for circle-ci builds"
- PR #1927[2975] - Using ninja for circle-ci builds
- PR #1926[2976] - Fixing serialization of std::array
- Issue #1925[2977] - Issues with static HPX libraries
- Issue #1924[2978] - Performance degrading over time
- Issue #1923[2979] - serialization of std::array appears broken in latest commit
- PR #1922[2980] - Container algorithms
- PR #1921[2981] - Tons of smaller quality improvements
- Issue #1920[2982] - Seg fault in hpx::serialization::output_archive::add_gid when running octotiger
- Issue #1919[2983] - Intel 15 compiler bug preventing HPX build
- PR #1918[2984] - Address sanitizer fixes
- PR #1917[2985] - Fixing compilation problems of parallel::sort with Intel compilers
- PR #1916[2986] - Making sure code compiles if HPX_WITH_HWLOC=Off
- Issue #1915[2987] - max_cores undefined if HPX_WITH_HWLOC=Off
- PR #1913[2988] - Add utility member functions for partitioned_vector
- PR #1912[2989] - Adding support for invoking actions to dataflow

[2967] https://github.com/STEllAR-GROUP/hpx/pull/1935
[2968] https://github.com/STEllAR-GROUP/hpx/pull/1934
[2969] https://github.com/STEllAR-GROUP/hpx/issues/1933
[2970] https://github.com/STEllAR-GROUP/hpx/pull/1932
[2971] https://github.com/STEllAR-GROUP/hpx/pull/1931
[2972] https://github.com/STEllAR-GROUP/hpx/issues/1930
[2973] https://github.com/STEllAR-GROUP/hpx/pull/1929
[2974] https://github.com/STEllAR-GROUP/hpx/pull/1928
[2975] https://github.com/STEllAR-GROUP/hpx/pull/1927
[2976] https://github.com/STEllAR-GROUP/hpx/pull/1926
[2977] https://github.com/STEllAR-GROUP/hpx/issues/1925
[2978] https://github.com/STEllAR-GROUP/hpx/issues/1924
[2979] https://github.com/STEllAR-GROUP/hpx/issues/1923
[2980] https://github.com/STEllAR-GROUP/hpx/pull/1922
[2981] https://github.com/STEllAR-GROUP/hpx/pull/1921
[2982] https://github.com/STEllAR-GROUP/hpx/issues/1920
[2983] https://github.com/STEllAR-GROUP/hpx/issues/1919
[2984] https://github.com/STEllAR-GROUP/hpx/pull/1918
[2985] https://github.com/STEllAR-GROUP/hpx/pull/1917
[2986] https://github.com/STEllAR-GROUP/hpx/pull/1916
[2987] https://github.com/STEllAR-GROUP/hpx/issues/1915
[2988] https://github.com/STEllAR-GROUP/hpx/pull/1913
[2989] https://github.com/STEllAR-GROUP/hpx/pull/1912

- PR #1911[2990] - Adding first batch of container algorithms
- PR #1910[2991] - Keep cmake_module_path
- PR #1909[2992] - Fix mpirun with pbs
- PR #1908[2993] - Changing parallel::sort to return the last iterator as proposed by N4560
- PR #1907[2994] - Adding a minimum version for Open MPI
- PR #1906[2995] - Updates to the Release Procedure
- PR #1905[2996] - Fixing #1903
- PR #1904[2997] - Making sure std containers are cleared before serialization loads data
- Issue #1903[2998] - When running octotiger, I get: assertion `'(*new_gids_)[gid].size() == 1' failed: HPX(assertion_failure)`
- Issue #1902[2999] - Immediate crash when running hpx/octotiger with _GLIBCXX_DEBUG defined.
- PR #1901[3000] - Making non-serializable classes non-serializable
- Issue #1900[3001] - Two possible issues with std::list serialization
- PR #1899[3002] - Fixing a problem with credit splitting as revealed by #1898
- Issue #1898[3003] - Accessing component from locality where it was not created segfaults
- PR #1897[3004] - Changing parallel::sort to return the last iterator as proposed by N4560
- Issue #1896[3005] - version 1.0?
- Issue #1895[3006] - Warning comment on numa_allocator is not very clear
- PR #1894[3007] - Add support for compilers that have thread_local
- PR #1893[3008] - Fixing 1890
- PR #1892[3009] - Adds typed future_type for executor_traits
- PR #1891[3010] - Fix wording in certain parallel algorithm docs
- Issue #1890[3011] - Invoking papi counters give segfault
- PR #1889[3012] - Fixing problems as reported by clang-check

[2990] https://github.com/STEllAR-GROUP/hpx/pull/1911
[2991] https://github.com/STEllAR-GROUP/hpx/pull/1910
[2992] https://github.com/STEllAR-GROUP/hpx/pull/1909
[2993] https://github.com/STEllAR-GROUP/hpx/pull/1908
[2994] https://github.com/STEllAR-GROUP/hpx/pull/1907
[2995] https://github.com/STEllAR-GROUP/hpx/pull/1906
[2996] https://github.com/STEllAR-GROUP/hpx/pull/1905
[2997] https://github.com/STEllAR-GROUP/hpx/pull/1904
[2998] https://github.com/STEllAR-GROUP/hpx/issues/1903
[2999] https://github.com/STEllAR-GROUP/hpx/issues/1902
[3000] https://github.com/STEllAR-GROUP/hpx/pull/1901
[3001] https://github.com/STEllAR-GROUP/hpx/issues/1900
[3002] https://github.com/STEllAR-GROUP/hpx/pull/1899
[3003] https://github.com/STEllAR-GROUP/hpx/issues/1898
[3004] https://github.com/STEllAR-GROUP/hpx/pull/1897
[3005] https://github.com/STEllAR-GROUP/hpx/issues/1896
[3006] https://github.com/STEllAR-GROUP/hpx/issues/1895
[3007] https://github.com/STEllAR-GROUP/hpx/pull/1894
[3008] https://github.com/STEllAR-GROUP/hpx/pull/1893
[3009] https://github.com/STEllAR-GROUP/hpx/pull/1892
[3010] https://github.com/STEllAR-GROUP/hpx/pull/1891
[3011] https://github.com/STEllAR-GROUP/hpx/issues/1890
[3012] https://github.com/STEllAR-GROUP/hpx/pull/1889

- PR #1888[3013] - WIP parallel is_heap
- PR #1887[3014] - Fixed resetting performance counters related to idle-rate, etc
- Issue #1886[3015] - Run hpx with qsub does not work
- PR #1885[3016] - Warning cleaning pass
- PR #1884[3017] - Add missing parallel algorithm header
- PR #1883[3018] - Add feature test for thread_local on Clang for TLS
- PR #1882[3019] - Fix some redundant qualifiers
- Issue #1881[3020] - Unable to compile Octotiger using HPX and Intel MPI on SuperMIC
- Issue #1880[3021] - clang with libc++ on Linux needs TLS case
- PR #1879[3022] - Doc fixes for #1868
- PR #1878[3023] - Simplify functions
- PR #1877[3024] - Removing most usage of Boost.Config
- PR #1876[3025] - Add missing parallel algorithms to algorithm.hpp
- PR #1875[3026] - Simplify callables
- PR #1874[3027] - Address long standing FIXME on using `std::unique_ptr` with incomplete types
- PR #1873[3028] - Fixing 1871
- PR #1872[3029] - Making sure PBS environment uses specified node list even if no PBS_NODEFILE env is available
- Issue #1871[3030] - Fortran checks should be optional
- PR #1870[3031] - Touch local::mutex
- PR #1869[3032] - Documentation refactoring based off #1868
- PR #1867[3033] - Embrace static_assert
- PR #1866[3034] - Fix #1803 with documentation refactoring
- PR #1865[3035] - Setting OUTPUT_NAME as target properties

[3013] https://github.com/STEllAR-GROUP/hpx/pull/1888
[3014] https://github.com/STEllAR-GROUP/hpx/pull/1887
[3015] https://github.com/STEllAR-GROUP/hpx/issues/1886
[3016] https://github.com/STEllAR-GROUP/hpx/pull/1885
[3017] https://github.com/STEllAR-GROUP/hpx/pull/1884
[3018] https://github.com/STEllAR-GROUP/hpx/pull/1883
[3019] https://github.com/STEllAR-GROUP/hpx/pull/1882
[3020] https://github.com/STEllAR-GROUP/hpx/issues/1881
[3021] https://github.com/STEllAR-GROUP/hpx/issues/1880
[3022] https://github.com/STEllAR-GROUP/hpx/pull/1879
[3023] https://github.com/STEllAR-GROUP/hpx/pull/1878
[3024] https://github.com/STEllAR-GROUP/hpx/pull/1877
[3025] https://github.com/STEllAR-GROUP/hpx/pull/1876
[3026] https://github.com/STEllAR-GROUP/hpx/pull/1875
[3027] https://github.com/STEllAR-GROUP/hpx/pull/1874
[3028] https://github.com/STEllAR-GROUP/hpx/pull/1873
[3029] https://github.com/STEllAR-GROUP/hpx/pull/1872
[3030] https://github.com/STEllAR-GROUP/hpx/issues/1871
[3031] https://github.com/STEllAR-GROUP/hpx/pull/1870
[3032] https://github.com/STEllAR-GROUP/hpx/pull/1869
[3033] https://github.com/STEllAR-GROUP/hpx/pull/1867
[3034] https://github.com/STEllAR-GROUP/hpx/pull/1866
[3035] https://github.com/STEllAR-GROUP/hpx/pull/1865

- PR #1863[3036] - Use SYSTEM for boost includes
- PR #1862[3037] - Minor cleanups
- PR #1861[3038] - Minor Corrections for Release
- PR #1860[3039] - Fixing hpx gdb script
- Issue #1859[3040] - reset_active_counters resets times and thread counts before some of the counters are evaluated
- PR #1858[3041] - Release V0.9.11
- PR #1857[3042] - removing diskperf example from 9.11 release
- PR #1856[3043] - fix return in packaged_task_base::reset()
- Issue #1842[3044] - Install error: file INSTALL cannot find libhpx_parcel_coalescing.so.0.9.11
- PR #1839[3045] - Adding fedora docs
- PR #1824[3046] - Changing version on master to V0.9.12
- PR #1818[3047] - Fixing #1748
- Issue #1815[3048] - seg fault in AGAS
- Issue #1803[3049] - wait_all documentation
- Issue #1796[3050] - Outdated documentation to be revised
- Issue #1759[3051] - glibc munmap_chunk or free(): invalid pointer on SuperMIC
- Issue #1753[3052] - HPX performance degrades with time since execution begins
- Issue #1748[3053] - All public HPX headers need to be self contained
- PR #1719[3054] - How to build HPX with Visual Studio
- Issue #1684[3055] - Race condition when using –hpx:connect?
- PR #1658[3056] - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1641[3057] - Generic client
- Issue #1632[3058] - heartbeat example fails on separate nodes

[3036] https://github.com/STEllAR-GROUP/hpx/pull/1863
[3037] https://github.com/STEllAR-GROUP/hpx/pull/1862
[3038] https://github.com/STEllAR-GROUP/hpx/pull/1861
[3039] https://github.com/STEllAR-GROUP/hpx/pull/1860
[3040] https://github.com/STEllAR-GROUP/hpx/issues/1859
[3041] https://github.com/STEllAR-GROUP/hpx/pull/1858
[3042] https://github.com/STEllAR-GROUP/hpx/pull/1857
[3043] https://github.com/STEllAR-GROUP/hpx/pull/1856
[3044] https://github.com/STEllAR-GROUP/hpx/issues/1842
[3045] https://github.com/STEllAR-GROUP/hpx/pull/1839
[3046] https://github.com/STEllAR-GROUP/hpx/pull/1824
[3047] https://github.com/STEllAR-GROUP/hpx/pull/1818
[3048] https://github.com/STEllAR-GROUP/hpx/issues/1815
[3049] https://github.com/STEllAR-GROUP/hpx/issues/1803
[3050] https://github.com/STEllAR-GROUP/hpx/issues/1796
[3051] https://github.com/STEllAR-GROUP/hpx/issues/1759
[3052] https://github.com/STEllAR-GROUP/hpx/issues/1753
[3053] https://github.com/STEllAR-GROUP/hpx/issues/1748
[3054] https://github.com/STEllAR-GROUP/hpx/pull/1719
[3055] https://github.com/STEllAR-GROUP/hpx/issues/1684
[3056] https://github.com/STEllAR-GROUP/hpx/pull/1658
[3057] https://github.com/STEllAR-GROUP/hpx/pull/1641
[3058] https://github.com/STEllAR-GROUP/hpx/issues/1632

- PR #1603[3059] - Adds preferred namespace check to inspect tool
- Issue #1559[3060] - Extend inspect tool
- Issue #1523[3061] - Remote async with deferred launch policy never executes
- Issue #1472[3062] - Serialization issues
- Issue #1457[3063] - Implement N4392: C++ Latches and Barriers
- PR #1444[3064] - Enabling usage of moveonly types for component construction
- Issue #1407[3065] - The Intel 13 compiler has failing unit tests
- Issue #1405[3066] - Allow component constructors to take movable only types
- Issue #1265[3067] - Enable dataflow() to be usable with actions
- Issue #1236[3068] - NUMA aware allocators
- Issue #802[3069] - Fix Broken Examples
- Issue #559[3070] - Add hpx::migrate facility
- Issue #449[3071] - Make actions with template arguments usable and add documentation
- Issue #279[3072] - Refactor addressing_service into a base class and two derived classes
- Issue #224[3073] - Changing thread state metadata is not thread safe
- Issue #55[3074] - Uniform syntax for enums should be implemented

2.11.11 *HPX* V0.9.11 (Nov 11, 2015)

Our main focus for this release was the design and development of a coherent set of higher-level APIs exposing various types of parallelism to the application programmer. We introduced the concepts of an `executor`, which can be used to customize the `where` and `when` of execution of tasks in the context of parallelizing codes. We extended all APIs related to managing parallel tasks to support executors which gives the user the choce of either using one of the predefined executor types or to provide its own, possibly application specific, executor. We paid very close attention to align all of these changes with the existing C++ Standards documents or with the ongoing proposals for standardization.

This release is the first after our change to a new development policy. We switched all development to be strictly performed on branches only, all direct commits to our main branch (`master`) are prohibited. Any change has to go through a peer review before it will be merged to `master`. As a result the overall stability of our code base has significantly increased, the development process itself has been simplified. This change manifests itself in a large number of pull-requests which have been merged (please see below for a full list of closed issues and pull-requests).

[3059] https://github.com/STEllAR-GROUP/hpx/pull/1603
[3060] https://github.com/STEllAR-GROUP/hpx/issues/1559
[3061] https://github.com/STEllAR-GROUP/hpx/issues/1523
[3062] https://github.com/STEllAR-GROUP/hpx/issues/1472
[3063] https://github.com/STEllAR-GROUP/hpx/issues/1457
[3064] https://github.com/STEllAR-GROUP/hpx/pull/1444
[3065] https://github.com/STEllAR-GROUP/hpx/issues/1407
[3066] https://github.com/STEllAR-GROUP/hpx/issues/1405
[3067] https://github.com/STEllAR-GROUP/hpx/issues/1265
[3068] https://github.com/STEllAR-GROUP/hpx/issues/1236
[3069] https://github.com/STEllAR-GROUP/hpx/issues/802
[3070] https://github.com/STEllAR-GROUP/hpx/issues/559
[3071] https://github.com/STEllAR-GROUP/hpx/issues/449
[3072] https://github.com/STEllAR-GROUP/hpx/issues/279
[3073] https://github.com/STEllAR-GROUP/hpx/issues/224
[3074] https://github.com/STEllAR-GROUP/hpx/issues/55

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 `hpx::components::component` and `hpx::components::component_base` have been added which currently just forward to the currently existing simple component facilities. The examples have been converted to only use those two classes.
- Added integration with the CircleCI[3075] hosted continuous integration service. This gives us constant and immediate feedback on the health of our master branch.
- The compiler configuration subsystem in the build system has been reimplemented. Instead of using Boost.Config we now use our own lightweight set of cmake scripts to determine the available language and library features supported by the used compiler.
- The API for creating instances of components has been consolidated. All component instances should be created using the `hpx::new_` only. It allows one to instantiate both, single component instances and multiple component instances. The placement of the created components can be controlled by special distribution policies. Please see the corresponding documentation outlining the use of `hpx::new_`.
- Introduced four new distribution policies which can be used with many API functions which traditionally expected to be used with a locality id. The new distribution policies are:
  - *hpx::components::default_distribution_policy* which tries to place multiple component instances as evenly as possible.
  - *hpx::components::colocating_distribution_policy* which will refer to the locality where a given component instance is currently placed.
  - *hpx::components::binpacking_distribution_policy* which will place multiple component instances as evenly as possible based on any performance counter.
  - `hpx::components::target_distribution_policy` which allows one to represent a given locality in the context of a distrwibution policy.
- The new distribution policies can now be also used with `hpx::async`. This change also deprecates `hpx::async_colocated(id, ...)` which now is replaced by a distribution policy: `hpx::async(hpx::colocated(id), ...)`.
- The `hpx::vector` and `hpx::unordered_map` data structures can now be used with the new distribution policies as well.
- The parallel facility `hpx::parallel::task_region` has been renamed to `hpx::parallel::task_block` based on the changes in the corresponding standardization proposal N4411[3076].
- 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[3077]).
- Added performance counters exposing data related to data transferred by input/output (filesystem) operations (thanks to Maciej Brodowicz).

[3075] https://circleci.com/gh/STEllAR-GROUP/hpx
[3076] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
[3077] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html

- 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/khuck/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[3078] 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.

[3078] https://github.com/STEllAR-GROUP/hpx/issues/1141

- 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[3079] 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::server::managed_compone` 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[3080] - Completely removing external/endian
- PR #1854[3081] - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1853[3082] - Updating CMake configuration to get correct version of TAU library
- PR #1852[3083] - Fixing Performance Problems with MPI Parcelport
- PR #1851[3084] - Fixing hpx_add_link_flag() and hpx_remove_link_flag()
- PR #1850[3085] - Fixing 1836, adding parallel::sort
- PR #1849[3086] - Fixing configuration for use of more than 64 cores
- PR #1848[3087] - Change default APEX version for release
- PR #1847[3088] - Fix client_base::then on release
- PR #1846[3089] - Removing broken lcos::local::channel from release
- PR #1845[3090] - Adding example demonstrating a possible safe-object implementation to release

[3079] http://thread.gmane.org/gmane.comp.lib.hpx.devel/196
[3080] https://github.com/STEllAR-GROUP/hpx/pull/1855
[3081] https://github.com/STEllAR-GROUP/hpx/pull/1854
[3082] https://github.com/STEllAR-GROUP/hpx/pull/1853
[3083] https://github.com/STEllAR-GROUP/hpx/pull/1852
[3084] https://github.com/STEllAR-GROUP/hpx/pull/1851
[3085] https://github.com/STEllAR-GROUP/hpx/pull/1850
[3086] https://github.com/STEllAR-GROUP/hpx/pull/1849
[3087] https://github.com/STEllAR-GROUP/hpx/pull/1848
[3088] https://github.com/STEllAR-GROUP/hpx/pull/1847
[3089] https://github.com/STEllAR-GROUP/hpx/pull/1846
[3090] https://github.com/STEllAR-GROUP/hpx/pull/1845

- PR #1844[3091] - Removing stubs from accumulator examples
- PR #1843[3092] - Don't pollute CMAKE_CXX_FLAGS through find_package()
- PR #1841[3093] - Fixing client_base<>::then
- PR #1840[3094] - Adding example demonstrating a possible safe-object implementation
- PR #1838[3095] - Update version rc1
- PR #1837[3096] - Removing broken lcos::local::channel
- PR #1835[3097] - Adding explicit move constructor and assignment operator to hpx::lcos::promise
- PR #1834[3098] - Making hpx::lcos::promise move-only
- PR #1833[3099] - Adding fedora docs
- Issue #1832[3100] - hpx::lcos::promise<> must be move-only
- PR #1831[3101] - Fixing resource manager gcc5.2
- PR #1830[3102] - Fix intel13
- PR #1829[3103] - Unbreaking thread test
- PR #1828[3104] - Fixing #1620
- PR #1827[3105] - Fixing a memory management issue for the Parquet application
- Issue #1826[3106] - Memory management issue in hpx::lcos::promise
- PR #1825[3107] - Adding hpx::components::component and hpx::components::component_base
- PR #1823[3108] - Adding git commit id to circleci build
- PR #1822[3109] - applying fixes suggested by clang 3.7
- PR #1821[3110] - Hyperlink fixes
- PR #1820[3111] - added parallel multi-locality sanity test
- PR #1819[3112] - Fixing #1667
- Issue #1817[3113] - Hyperlinks generated by inspect tool are wrong

[3091] https://github.com/STEllAR-GROUP/hpx/pull/1844
[3092] https://github.com/STEllAR-GROUP/hpx/pull/1843
[3093] https://github.com/STEllAR-GROUP/hpx/pull/1841
[3094] https://github.com/STEllAR-GROUP/hpx/pull/1840
[3095] https://github.com/STEllAR-GROUP/hpx/pull/1838
[3096] https://github.com/STEllAR-GROUP/hpx/pull/1837
[3097] https://github.com/STEllAR-GROUP/hpx/pull/1835
[3098] https://github.com/STEllAR-GROUP/hpx/pull/1834
[3099] https://github.com/STEllAR-GROUP/hpx/pull/1833
[3100] https://github.com/STEllAR-GROUP/hpx/issues/1832
[3101] https://github.com/STEllAR-GROUP/hpx/pull/1831
[3102] https://github.com/STEllAR-GROUP/hpx/pull/1830
[3103] https://github.com/STEllAR-GROUP/hpx/pull/1829
[3104] https://github.com/STEllAR-GROUP/hpx/pull/1828
[3105] https://github.com/STEllAR-GROUP/hpx/pull/1827
[3106] https://github.com/STEllAR-GROUP/hpx/issues/1826
[3107] https://github.com/STEllAR-GROUP/hpx/pull/1825
[3108] https://github.com/STEllAR-GROUP/hpx/pull/1823
[3109] https://github.com/STEllAR-GROUP/hpx/pull/1822
[3110] https://github.com/STEllAR-GROUP/hpx/pull/1821
[3111] https://github.com/STEllAR-GROUP/hpx/pull/1820
[3112] https://github.com/STEllAR-GROUP/hpx/pull/1819
[3113] https://github.com/STEllAR-GROUP/hpx/issues/1817

- PR #1816[3114] - Support hpxrx
- PR #1814[3115] - Fix async to dispatch to the correct locality in all cases
- Issue #1813[3116] - async(launch::..., action(), ...) always invokes locally
- PR #1812[3117] - fixed syntax error in CMakeLists.txt
- PR #1811[3118] - Agas optimizations
- PR #1810[3119] - drop superfluous typedefs
- PR #1809[3120] - Allow HPX to be used as an optional package in 3rd party code
- PR #1808[3121] - Fixing #1723
- PR #1807[3122] - Making sure resolve_localities does not hang during normal operation
- Issue #1806[3123] - Spinlock no longer movable and deletes operator '=', breaks MiniGhost
- Issue #1804[3124] - register_with_basename causes hangs
- PR #1801[3125] - Enhanced the inspect tool to take user directly to the problem with hyperlinks
- Issue #1800[3126] - Problems compiling application on smic
- PR #1799[3127] - Fixing cv exceptions
- PR #1798[3128] - Documentation refactoring & updating
- PR #1797[3129] - Updating the activeharmony CMake module
- PR #1795[3130] - Fixing cv
- PR #1794[3131] - Fix connect with hpx::runtime_mode_connect
- PR #1793[3132] - fix a wrong use of HPX_MAX_CPU_COUNT instead of HPX_HAVE_MAX_CPU_COUNT
- PR #1792[3133] - Allow for default constructed parcel instances to be moved
- PR #1791[3134] - Fix connect with hpx::runtime_mode_connect
- Issue #1790[3135] - assertion `action_.get()` failed: HPX(assertion_failure) when running Octotiger with pull request 1786
- PR #1789[3136] - Fixing discover_counter_types API function

[3114] https://github.com/STEllAR-GROUP/hpx/pull/1816
[3115] https://github.com/STEllAR-GROUP/hpx/pull/1814
[3116] https://github.com/STEllAR-GROUP/hpx/issues/1813
[3117] https://github.com/STEllAR-GROUP/hpx/pull/1812
[3118] https://github.com/STEllAR-GROUP/hpx/pull/1811
[3119] https://github.com/STEllAR-GROUP/hpx/pull/1810
[3120] https://github.com/STEllAR-GROUP/hpx/pull/1809
[3121] https://github.com/STEllAR-GROUP/hpx/pull/1808
[3122] https://github.com/STEllAR-GROUP/hpx/pull/1807
[3123] https://github.com/STEllAR-GROUP/hpx/issues/1806
[3124] https://github.com/STEllAR-GROUP/hpx/issues/1804
[3125] https://github.com/STEllAR-GROUP/hpx/pull/1801
[3126] https://github.com/STEllAR-GROUP/hpx/issues/1800
[3127] https://github.com/STEllAR-GROUP/hpx/pull/1799
[3128] https://github.com/STEllAR-GROUP/hpx/pull/1798
[3129] https://github.com/STEllAR-GROUP/hpx/pull/1797
[3130] https://github.com/STEllAR-GROUP/hpx/pull/1795
[3131] https://github.com/STEllAR-GROUP/hpx/pull/1794
[3132] https://github.com/STEllAR-GROUP/hpx/pull/1793
[3133] https://github.com/STEllAR-GROUP/hpx/pull/1792
[3134] https://github.com/STEllAR-GROUP/hpx/pull/1791
[3135] https://github.com/STEllAR-GROUP/hpx/issues/1790
[3136] https://github.com/STEllAR-GROUP/hpx/pull/1789

- Issue #1788[3137] - connect with hpx::runtime_mode_connect
- Issue #1787[3138] - discover_counter_types not working
- PR #1786[3139] - Changing addressing_service to use std::unordered_map instead of std::map
- PR #1785[3140] - Fix is_iterator for container algorithms
- PR #1784[3141] - Adding new command line options:
- PR #1783[3142] - Minor changes for APEX support
- PR #1782[3143] - Drop legacy forwarding action traits
- PR #1781[3144] - Attempt to resolve the race between cv::wait_xxx and cv::notify_all
- PR #1780[3145] - Removing serialize_sequence
- PR #1779[3146] - Fixed #1501: hwloc configuration options are wrong for MIC
- PR #1778[3147] - Removing ability to enable/disable parcel handling
- PR #1777[3148] - Completely removing stackless threads
- PR #1776[3149] - Cleaning up util/plugin
- PR #1775[3150] - Agas fixes
- PR #1774[3151] - Action invocation count
- PR #1773[3152] - replaced MSVC variable with WIN32
- PR #1772[3153] - Fixing Problems in MPI parcelport and future serialization.
- PR #1771[3154] - Fixing intel 13 compiler errors related to variadic template template parameters for `lcos::when_` tests
- PR #1770[3155] - Forwarding decay to `std::`
- PR #1769[3156] - Add more characters with special regex meaning to the existing patch
- PR #1768[3157] - Adding test for receive_buffer
- PR #1767[3158] - Making sure that uptime counter throws exception on any attempt to be reset
- PR #1766[3159] - Cleaning up code related to throttling scheduler

3137 https://github.com/STEllAR-GROUP/hpx/issues/1788
3138 https://github.com/STEllAR-GROUP/hpx/issues/1787
3139 https://github.com/STEllAR-GROUP/hpx/pull/1786
3140 https://github.com/STEllAR-GROUP/hpx/pull/1785
3141 https://github.com/STEllAR-GROUP/hpx/pull/1784
3142 https://github.com/STEllAR-GROUP/hpx/pull/1783
3143 https://github.com/STEllAR-GROUP/hpx/pull/1782
3144 https://github.com/STEllAR-GROUP/hpx/pull/1781
3145 https://github.com/STEllAR-GROUP/hpx/pull/1780
3146 https://github.com/STEllAR-GROUP/hpx/pull/1779
3147 https://github.com/STEllAR-GROUP/hpx/pull/1778
3148 https://github.com/STEllAR-GROUP/hpx/pull/1777
3149 https://github.com/STEllAR-GROUP/hpx/pull/1776
3150 https://github.com/STEllAR-GROUP/hpx/pull/1775
3151 https://github.com/STEllAR-GROUP/hpx/pull/1774
3152 https://github.com/STEllAR-GROUP/hpx/pull/1773
3153 https://github.com/STEllAR-GROUP/hpx/pull/1772
3154 https://github.com/STEllAR-GROUP/hpx/pull/1771
3155 https://github.com/STEllAR-GROUP/hpx/pull/1770
3156 https://github.com/STEllAR-GROUP/hpx/pull/1769
3157 https://github.com/STEllAR-GROUP/hpx/pull/1768
3158 https://github.com/STEllAR-GROUP/hpx/pull/1767
3159 https://github.com/STEllAR-GROUP/hpx/pull/1766

- PR #1765[3160] - Restricting thread_data to creating only with intrusive_pointers
- PR #1764[3161] - Fixing 1763
- Issue #1763[3162] - UB in thread_data::operator delete
- PR #1762[3163] - Making sure all serialization registries/factories are unique
- PR #1761[3164] - Fixed #1751: hpx::future::wait_for fails a simple test
- PR #1758[3165] - Fixing #1757
- Issue #1757[3166] - pinning not correct using –hpx:bind
- Issue #1756[3167] - compilation error with MinGW
- PR #1755[3168] - Making output serialization const-correct
- Issue #1753[3169] - HPX performance degrades with time since execution begins
- Issue #1752[3170] - Error in AGAS
- Issue #1751[3171] - hpx::future::wait_for fails a simple test
- PR #1750[3172] - Removing hpx_fwd.hpp includes
- PR #1749[3173] - Simplify result_of and friends
- PR #1747[3174] - Removed superfluous code from message_buffer.hpp
- PR #1746[3175] - Tuple dependencies
- Issue #1745[3176] - Broken when_some which takes iterators
- PR #1744[3177] - Refining archive interface
- PR #1743[3178] - Fixing when_all when only a single future is passed
- PR #1742[3179] - Config includes
- PR #1741[3180] - Os executors
- Issue #1740[3181] - hpx::promise has some problems
- PR #1739[3182] - Parallel composition with generic containers

3160 https://github.com/STEllAR-GROUP/hpx/pull/1765
3161 https://github.com/STEllAR-GROUP/hpx/pull/1764
3162 https://github.com/STEllAR-GROUP/hpx/issues/1763
3163 https://github.com/STEllAR-GROUP/hpx/pull/1762
3164 https://github.com/STEllAR-GROUP/hpx/pull/1761
3165 https://github.com/STEllAR-GROUP/hpx/pull/1758
3166 https://github.com/STEllAR-GROUP/hpx/issues/1757
3167 https://github.com/STEllAR-GROUP/hpx/issues/1756
3168 https://github.com/STEllAR-GROUP/hpx/pull/1755
3169 https://github.com/STEllAR-GROUP/hpx/issues/1753
3170 https://github.com/STEllAR-GROUP/hpx/issues/1752
3171 https://github.com/STEllAR-GROUP/hpx/issues/1751
3172 https://github.com/STEllAR-GROUP/hpx/pull/1750
3173 https://github.com/STEllAR-GROUP/hpx/pull/1749
3174 https://github.com/STEllAR-GROUP/hpx/pull/1747
3175 https://github.com/STEllAR-GROUP/hpx/pull/1746
3176 https://github.com/STEllAR-GROUP/hpx/issues/1745
3177 https://github.com/STEllAR-GROUP/hpx/pull/1744
3178 https://github.com/STEllAR-GROUP/hpx/pull/1743
3179 https://github.com/STEllAR-GROUP/hpx/pull/1742
3180 https://github.com/STEllAR-GROUP/hpx/pull/1741
3181 https://github.com/STEllAR-GROUP/hpx/issues/1740
3182 https://github.com/STEllAR-GROUP/hpx/pull/1739

- Issue #1738[3183] - After building program and successfully linking to a version of hpx DHPX_DIR seems to be ignored
- Issue #1737[3184] - Uptime problems
- PR #1736[3185] - added convenience c-tor and begin()/end() to serialize_buffer
- PR #1735[3186] - Config includes
- PR #1734[3187] - Fixed #1688: Add timer counters for tfunc_total and exec_total
- Issue #1733[3188] - Add unit test for hpx/lcos/local/receive_buffer.hpp
- PR #1732[3189] - Renaming get_os_thread_count
- PR #1731[3190] - Basename registration
- Issue #1730[3191] - Use after move of thread_init_data
- PR #1729[3192] - Rewriting channel based on new gate component
- PR #1728[3193] - Fixing #1722
- PR #1727[3194] - Fixing compile problems with apply_colocated
- PR #1726[3195] - Apex integration
- PR #1725[3196] - fixed test timeouts
- PR #1724[3197] - Renaming vector
- Issue #1723[3198] - Drop support for intel compilers and gcc 4.4. based standard libs
- Issue #1722[3199] - Add support for detecting non-ready futures before serialization
- PR #1721[3200] - Unifying parallel executors, initializing from launch policy
- PR #1720[3201] - dropped superfluous typedef
- Issue #1718[3202] - Windows 10 x64, VS 2015 - Unknown CMake command "add_hpx_pseudo_target".
- PR #1717[3203] - Timed executor traits for thread-executors
- PR #1716[3204] - serialization of arrays didn't work with non-pod types. fixed
- PR #1715[3205] - List serialization

3183 https://github.com/STEllAR-GROUP/hpx/issues/1738
3184 https://github.com/STEllAR-GROUP/hpx/issues/1737
3185 https://github.com/STEllAR-GROUP/hpx/pull/1736
3186 https://github.com/STEllAR-GROUP/hpx/pull/1735
3187 https://github.com/STEllAR-GROUP/hpx/pull/1734
3188 https://github.com/STEllAR-GROUP/hpx/issues/1733
3189 https://github.com/STEllAR-GROUP/hpx/pull/1732
3190 https://github.com/STEllAR-GROUP/hpx/pull/1731
3191 https://github.com/STEllAR-GROUP/hpx/issues/1730
3192 https://github.com/STEllAR-GROUP/hpx/pull/1729
3193 https://github.com/STEllAR-GROUP/hpx/pull/1728
3194 https://github.com/STEllAR-GROUP/hpx/pull/1727
3195 https://github.com/STEllAR-GROUP/hpx/pull/1726
3196 https://github.com/STEllAR-GROUP/hpx/pull/1725
3197 https://github.com/STEllAR-GROUP/hpx/pull/1724
3198 https://github.com/STEllAR-GROUP/hpx/issues/1723
3199 https://github.com/STEllAR-GROUP/hpx/issues/1722
3200 https://github.com/STEllAR-GROUP/hpx/pull/1721
3201 https://github.com/STEllAR-GROUP/hpx/pull/1720
3202 https://github.com/STEllAR-GROUP/hpx/issues/1718
3203 https://github.com/STEllAR-GROUP/hpx/pull/1717
3204 https://github.com/STEllAR-GROUP/hpx/pull/1716
3205 https://github.com/STEllAR-GROUP/hpx/pull/1715

- PR #1714[3206] - changing misspellings
- PR #1713[3207] - Fixed distribution policy executors
- PR #1712[3208] - Moving library detection to be executed after feature tests
- PR #1711[3209] - Simplify parcel
- PR #1710[3210] - Compile only tests
- PR #1709[3211] - Implemented timed executors
- PR #1708[3212] - Implement parallel::executor_traits for thread-executors
- PR #1707[3213] - Various fixes to threads::executors to make custom schedulers work
- PR #1706[3214] - Command line option –hpx:cores does not work as expected
- Issue #1705[3215] - command line option –hpx:cores does not work as expected
- PR #1704[3216] - vector deserialization is speeded up a little
- PR #1703[3217] - Fixing shared_mutes
- Issue #1702[3218] - Shared_mutex does not compile with no_mutex cond_var
- PR #1701[3219] - Add distribution_policy_executor
- PR #1700[3220] - Executor parameters
- PR #1699[3221] - Readers writer lock
- PR #1698[3222] - Remove leftovers
- PR #1697[3223] - Fixing held locks
- PR #1696[3224] - Modified Scan Partitioner for Algorithms
- PR #1695[3225] - This thread executors
- PR #1694[3226] - Fixed #1688: Add timer counters for tfunc_total and exec_total
- PR #1693[3227] - Fix #1691: is_executor template specification fails for inherited executors
- PR #1692[3228] - Fixed #1662: Possible exception source in coalescing_message_handler

3206 https://github.com/STEllAR-GROUP/hpx/pull/1714
3207 https://github.com/STEllAR-GROUP/hpx/pull/1713
3208 https://github.com/STEllAR-GROUP/hpx/pull/1712
3209 https://github.com/STEllAR-GROUP/hpx/pull/1711
3210 https://github.com/STEllAR-GROUP/hpx/pull/1710
3211 https://github.com/STEllAR-GROUP/hpx/pull/1709
3212 https://github.com/STEllAR-GROUP/hpx/pull/1708
3213 https://github.com/STEllAR-GROUP/hpx/pull/1707
3214 https://github.com/STEllAR-GROUP/hpx/pull/1706
3215 https://github.com/STEllAR-GROUP/hpx/issues/1705
3216 https://github.com/STEllAR-GROUP/hpx/pull/1704
3217 https://github.com/STEllAR-GROUP/hpx/pull/1703
3218 https://github.com/STEllAR-GROUP/hpx/issues/1702
3219 https://github.com/STEllAR-GROUP/hpx/pull/1701
3220 https://github.com/STEllAR-GROUP/hpx/pull/1700
3221 https://github.com/STEllAR-GROUP/hpx/pull/1699
3222 https://github.com/STEllAR-GROUP/hpx/pull/1698
3223 https://github.com/STEllAR-GROUP/hpx/pull/1697
3224 https://github.com/STEllAR-GROUP/hpx/pull/1696
3225 https://github.com/STEllAR-GROUP/hpx/pull/1695
3226 https://github.com/STEllAR-GROUP/hpx/pull/1694
3227 https://github.com/STEllAR-GROUP/hpx/pull/1693
3228 https://github.com/STEllAR-GROUP/hpx/pull/1692

- Issue #1691[3229] - is_executor template specification fails for inherited executors
- PR #1690[3230] - added macro for non-intrusive serialization of classes without a default c-tor
- PR #1689[3231] - Replace value_or_error with custom storage, unify future_data state
- Issue #1688[3232] - Add timer counters for tfunc_total and exec_total
- PR #1687[3233] - Fixed interval timer
- PR #1686[3234] - Fixing cmake warnings about not existing pseudo target dependencies
- PR #1685[3235] - Converting partitioners to use bulk async execute
- PR #1683[3236] - Adds a tool for inspect that checks for character limits
- PR #1682[3237] - Change project name to (uppercase) HPX
- PR #1681[3238] - Counter shortnames
- PR #1680[3239] - Extended Non-intrusive Serialization to Ease Usage for Library Developers
- PR #1679[3240] - Working on 1544: More executor changes
- PR #1678[3241] - Transpose fixes
- PR #1677[3242] - Improve Boost compatibility check
- PR #1676[3243] - 1d stencil fix
- Issue #1675[3244] - hpx project name is not HPX
- PR #1674[3245] - Fixing the MPI parcelport
- PR #1673[3246] - added move semantics to map/vector deserialization
- PR #1672[3247] - Vs2015 await
- PR #1671[3248] - Adapt transform for #1668
- PR #1670[3249] - Started to work on #1668
- PR #1669[3250] - Add this_thread_executors
- Issue #1667[3251] - Apple build instructions in docs are out of date

[3229] https://github.com/STEllAR-GROUP/hpx/issues/1691
[3230] https://github.com/STEllAR-GROUP/hpx/pull/1690
[3231] https://github.com/STEllAR-GROUP/hpx/pull/1689
[3232] https://github.com/STEllAR-GROUP/hpx/issues/1688
[3233] https://github.com/STEllAR-GROUP/hpx/pull/1687
[3234] https://github.com/STEllAR-GROUP/hpx/pull/1686
[3235] https://github.com/STEllAR-GROUP/hpx/pull/1685
[3236] https://github.com/STEllAR-GROUP/hpx/pull/1683
[3237] https://github.com/STEllAR-GROUP/hpx/pull/1682
[3238] https://github.com/STEllAR-GROUP/hpx/pull/1681
[3239] https://github.com/STEllAR-GROUP/hpx/pull/1680
[3240] https://github.com/STEllAR-GROUP/hpx/pull/1679
[3241] https://github.com/STEllAR-GROUP/hpx/pull/1678
[3242] https://github.com/STEllAR-GROUP/hpx/pull/1677
[3243] https://github.com/STEllAR-GROUP/hpx/pull/1676
[3244] https://github.com/STEllAR-GROUP/hpx/issues/1675
[3245] https://github.com/STEllAR-GROUP/hpx/pull/1674
[3246] https://github.com/STEllAR-GROUP/hpx/pull/1673
[3247] https://github.com/STEllAR-GROUP/hpx/pull/1672
[3248] https://github.com/STEllAR-GROUP/hpx/pull/1671
[3249] https://github.com/STEllAR-GROUP/hpx/pull/1670
[3250] https://github.com/STEllAR-GROUP/hpx/pull/1669
[3251] https://github.com/STEllAR-GROUP/hpx/issues/1667

- PR #1666[3252] - Apex integration
- PR #1665[3253] - Fixes an error with the whitespace check that showed the incorrect location of the error
- Issue #1664[3254] - Inspect tool found incorrect endline whitespace
- PR #1663[3255] - Improve use of locks
- Issue #1662[3256] - Possible exception source in coalescing_message_handler
- PR #1661[3257] - Added support for 128bit number serialization
- PR #1660[3258] - Serialization 128bits
- PR #1659[3259] - Implemented inner_product and adjacent_diff algos
- PR #1658[3260] - Add serialization for std::set (as there is for std::vector and std::map)
- PR #1657[3261] - Use of shared_ptr in io_service_pool changed to unique_ptr
- Issue #1656[3262] - 1d_stencil codes all have wrong factor
- PR #1654[3263] - When using runtime_mode_connect, find the correct localhost public ip address
- PR #1653[3264] - Fixing 1617
- PR #1652[3265] - Remove traits::action_may_require_id_splitting
- PR #1651[3266] - Fixed performance counters related to AGAS cache timings
- PR #1650[3267] - Remove leftovers of traits::type_size
- PR #1649[3268] - Shorten target names on Windows to shorten used path names
- PR #1648[3269] - Fixing problems introduced by merging #1623 for older compilers
- PR #1647[3270] - Simplify running automatic builds on Windows
- Issue #1646[3271] - Cache insert and update performance counters are broken
- Issue #1644[3272] - Remove leftovers of traits::type_size
- Issue #1643[3273] - Remove traits::action_may_require_id_splitting
- PR #1642[3274] - Adds spell checker to the inspect tool for qbk and doxygen comments

[3252] https://github.com/STEllAR-GROUP/hpx/pull/1666
[3253] https://github.com/STEllAR-GROUP/hpx/pull/1665
[3254] https://github.com/STEllAR-GROUP/hpx/issues/1664
[3255] https://github.com/STEllAR-GROUP/hpx/pull/1663
[3256] https://github.com/STEllAR-GROUP/hpx/issues/1662
[3257] https://github.com/STEllAR-GROUP/hpx/pull/1661
[3258] https://github.com/STEllAR-GROUP/hpx/pull/1660
[3259] https://github.com/STEllAR-GROUP/hpx/pull/1659
[3260] https://github.com/STEllAR-GROUP/hpx/pull/1658
[3261] https://github.com/STEllAR-GROUP/hpx/pull/1657
[3262] https://github.com/STEllAR-GROUP/hpx/issues/1656
[3263] https://github.com/STEllAR-GROUP/hpx/pull/1654
[3264] https://github.com/STEllAR-GROUP/hpx/pull/1653
[3265] https://github.com/STEllAR-GROUP/hpx/pull/1652
[3266] https://github.com/STEllAR-GROUP/hpx/pull/1651
[3267] https://github.com/STEllAR-GROUP/hpx/pull/1650
[3268] https://github.com/STEllAR-GROUP/hpx/pull/1649
[3269] https://github.com/STEllAR-GROUP/hpx/pull/1648
[3270] https://github.com/STEllAR-GROUP/hpx/pull/1647
[3271] https://github.com/STEllAR-GROUP/hpx/issues/1646
[3272] https://github.com/STEllAR-GROUP/hpx/issues/1644
[3273] https://github.com/STEllAR-GROUP/hpx/issues/1643
[3274] https://github.com/STEllAR-GROUP/hpx/pull/1642

- PR #1640[3275] - First step towards fixing 688
- PR #1639[3276] - Re-apply remaining changes from limit_dataflow_recursion branch
- PR #1638[3277] - This fixes possible deadlock in the test ignore_while_locked_1485
- PR #1637[3278] - Fixing hpx::wait_all() invoked with two vector<future<T>>
- PR #1636[3279] - Partially re-apply changes from limit_dataflow_recursion branch
- PR #1635[3280] - Adding missing test for #1572
- PR #1634[3281] - Revert "Limit recursion-depth in dataflow to a configurable constant"
- PR #1633[3282] - Add command line option to ignore batch environment
- PR #1631[3283] - hpx::lcos::queue exhibits strange behavior
- PR #1630[3284] - Fixed endline_whitespace_check.cpp to detect lines with only whitespace
- Issue #1629[3285] - Inspect trailing whitespace checker problem
- PR #1628[3286] - Removed meaningless const qualifiers. Minor icpc fix.
- PR #1627[3287] - Fixing the queue LCO and add example demonstrating its use
- PR #1626[3288] - Deprecating get_gid(), add get_id() and get_unmanaged_id()
- PR #1625[3289] - Allowing to specify whether to send credits along with message
- Issue #1624[3290] - Lifetime issue
- Issue #1623[3291] - hpx::wait_all() invoked with two vector<future<T>> fails
- PR #1622[3292] - Executor partitioners
- PR #1621[3293] - Clean up coroutines implementation
- Issue #1620[3294] - Revert #1535
- PR #1619[3295] - Fix result type calculation for hpx::make_continuation
- PR #1618[3296] - Fixing RDTSC on Xeon/Phi
- Issue #1617[3297] - hpx cmake not working when run as a subproject

3275 https://github.com/STEllAR-GROUP/hpx/pull/1640
3276 https://github.com/STEllAR-GROUP/hpx/pull/1639
3277 https://github.com/STEllAR-GROUP/hpx/pull/1638
3278 https://github.com/STEllAR-GROUP/hpx/pull/1637
3279 https://github.com/STEllAR-GROUP/hpx/pull/1636
3280 https://github.com/STEllAR-GROUP/hpx/pull/1635
3281 https://github.com/STEllAR-GROUP/hpx/pull/1634
3282 https://github.com/STEllAR-GROUP/hpx/pull/1633
3283 https://github.com/STEllAR-GROUP/hpx/pull/1631
3284 https://github.com/STEllAR-GROUP/hpx/pull/1630
3285 https://github.com/STEllAR-GROUP/hpx/issues/1629
3286 https://github.com/STEllAR-GROUP/hpx/pull/1628
3287 https://github.com/STEllAR-GROUP/hpx/pull/1627
3288 https://github.com/STEllAR-GROUP/hpx/pull/1626
3289 https://github.com/STEllAR-GROUP/hpx/pull/1625
3290 https://github.com/STEllAR-GROUP/hpx/issues/1624
3291 https://github.com/STEllAR-GROUP/hpx/issues/1623
3292 https://github.com/STEllAR-GROUP/hpx/pull/1622
3293 https://github.com/STEllAR-GROUP/hpx/pull/1621
3294 https://github.com/STEllAR-GROUP/hpx/issues/1620
3295 https://github.com/STEllAR-GROUP/hpx/pull/1619
3296 https://github.com/STEllAR-GROUP/hpx/pull/1618
3297 https://github.com/STEllAR-GROUP/hpx/issues/1617

- Issue #1616[3298] - cmake problem resulting in RDTSC not working correctly for Xeon Phi creates very strange results for duration counters
- Issue #1615[3299] - hpx::make_continuation requires input and output to be the same
- PR #1614[3300] - Fixed remove copy test
- Issue #1613[3301] - Dataflow causes stack overflow
- PR #1612[3302] - Modified foreach partitioner to use bulk execute
- PR #1611[3303] - Limit recursion-depth in dataflow to a configurable constant
- PR #1610[3304] - Increase timeout for CircleCI
- PR #1609[3305] - Refactoring thread manager, mainly extracting thread pool
- PR #1608[3306] - Fixed running multiple localities without localities parameter
- PR #1607[3307] - More algorithm fixes to adjacentfind
- Issue #1606[3308] - Running without localities parameter binds to bogus port range
- Issue #1605[3309] - Too many serializations
- PR #1604[3310] - Changes the HPX image into a hyperlink
- PR #1601[3311] - Fixing problems with remove_copy algorithm tests
- PR #1600[3312] - Actions with ids cleanup
- PR #1599[3313] - Duplicate binding of global ids should fail
- PR #1598[3314] - Fixing array access
- PR #1597[3315] - Improved the reliability of connecting/disconnecting localities
- Issue #1596[3316] - Duplicate id binding should fail
- PR #1595[3317] - Fixing more cmake config constants
- PR #1594[3318] - Fixing preprocessor constant used to enable C++11 chrono
- PR #1593[3319] - Adding operator|() for hpx::launch
- Issue #1592[3320] - Error (typo) in the docs

[3298] https://github.com/STEllAR-GROUP/hpx/issues/1616
[3299] https://github.com/STEllAR-GROUP/hpx/issues/1615
[3300] https://github.com/STEllAR-GROUP/hpx/pull/1614
[3301] https://github.com/STEllAR-GROUP/hpx/issues/1613
[3302] https://github.com/STEllAR-GROUP/hpx/pull/1612
[3303] https://github.com/STEllAR-GROUP/hpx/pull/1611
[3304] https://github.com/STEllAR-GROUP/hpx/pull/1610
[3305] https://github.com/STEllAR-GROUP/hpx/pull/1609
[3306] https://github.com/STEllAR-GROUP/hpx/pull/1608
[3307] https://github.com/STEllAR-GROUP/hpx/pull/1607
[3308] https://github.com/STEllAR-GROUP/hpx/issues/1606
[3309] https://github.com/STEllAR-GROUP/hpx/issues/1605
[3310] https://github.com/STEllAR-GROUP/hpx/pull/1604
[3311] https://github.com/STEllAR-GROUP/hpx/pull/1601
[3312] https://github.com/STEllAR-GROUP/hpx/pull/1600
[3313] https://github.com/STEllAR-GROUP/hpx/pull/1599
[3314] https://github.com/STEllAR-GROUP/hpx/pull/1598
[3315] https://github.com/STEllAR-GROUP/hpx/pull/1597
[3316] https://github.com/STEllAR-GROUP/hpx/issues/1596
[3317] https://github.com/STEllAR-GROUP/hpx/pull/1595
[3318] https://github.com/STEllAR-GROUP/hpx/pull/1594
[3319] https://github.com/STEllAR-GROUP/hpx/pull/1593
[3320] https://github.com/STEllAR-GROUP/hpx/issues/1592

- Issue #1590[3321] - CMake fails when CMAKE_BINARY_DIR contains '+'.
- Issue #1589[3322] - Disconnecting a locality results in segfault using heartbeat example
- PR #1588[3323] - Fix doc string for config option HPX_WITH_EXAMPLES
- PR #1586[3324] - Fixing 1493
- PR #1585[3325] - Additional Check for Inspect Tool to detect Endline Whitespace
- Issue #1584[3326] - Clean up coroutines implementation
- PR #1583[3327] - Adding a check for end line whitespace
- PR #1582[3328] - Attempt to fix assert firing after scheduling loop was exited
- PR #1581[3329] - Fixed adjacentfind_binary test
- PR #1580[3330] - Prevent some of the internal cmake lists from growing indefinitely
- PR #1579[3331] - Removing type_size trait, replacing it with special archive type
- Issue #1578[3332] - Remove demangle_helper
- PR #1577[3333] - Get ptr problems
- Issue #1576[3334] - Refactor async, dataflow, and future::then
- PR #1575[3335] - Fixing tests for parallel rotate
- PR #1574[3336] - Cleaning up schedulers
- PR #1573[3337] - Fixing thread pool executor
- PR #1572[3338] - Fixing number of configured localities
- PR #1571[3339] - Reimplement decay
- PR #1570[3340] - Refactoring async, apply, and dataflow APIs
- PR #1569[3341] - Changed range for mach-o library lookup
- PR #1568[3342] - Mark decltype support as required
- PR #1567[3343] - Removed const from algorithms

[3321] https://github.com/STEllAR-GROUP/hpx/issues/1590
[3322] https://github.com/STEllAR-GROUP/hpx/issues/1589
[3323] https://github.com/STEllAR-GROUP/hpx/pull/1588
[3324] https://github.com/STEllAR-GROUP/hpx/pull/1586
[3325] https://github.com/STEllAR-GROUP/hpx/pull/1585
[3326] https://github.com/STEllAR-GROUP/hpx/issues/1584
[3327] https://github.com/STEllAR-GROUP/hpx/pull/1583
[3328] https://github.com/STEllAR-GROUP/hpx/pull/1582
[3329] https://github.com/STEllAR-GROUP/hpx/pull/1581
[3330] https://github.com/STEllAR-GROUP/hpx/pull/1580
[3331] https://github.com/STEllAR-GROUP/hpx/pull/1579
[3332] https://github.com/STEllAR-GROUP/hpx/issues/1578
[3333] https://github.com/STEllAR-GROUP/hpx/pull/1577
[3334] https://github.com/STEllAR-GROUP/hpx/issues/1576
[3335] https://github.com/STEllAR-GROUP/hpx/pull/1575
[3336] https://github.com/STEllAR-GROUP/hpx/pull/1574
[3337] https://github.com/STEllAR-GROUP/hpx/pull/1573
[3338] https://github.com/STEllAR-GROUP/hpx/pull/1572
[3339] https://github.com/STEllAR-GROUP/hpx/pull/1571
[3340] https://github.com/STEllAR-GROUP/hpx/pull/1570
[3341] https://github.com/STEllAR-GROUP/hpx/pull/1569
[3342] https://github.com/STEllAR-GROUP/hpx/pull/1568
[3343] https://github.com/STEllAR-GROUP/hpx/pull/1567

- Issue #1566[3344] - CMAKE Configuration Test Failures for clang 3.5 on debian
- PR #1565[3345] - Dylib support
- PR #1564[3346] - Converted partitioners and some algorithms to use executors
- PR #1563[3347] - Fix several #includes for Boost.Preprocessor
- PR #1562[3348] - Adding configuration option disabling/enabling all message handlers
- PR #1561[3349] - Removed all occurrences of boost::move replacing it with std::move
- Issue #1560[3350] - Leftover HPX_REGISTER_ACTION_DECLARATION_2
- PR #1558[3351] - Revisit async/apply SFINAE conditions
- PR #1557[3352] - Removing type_size trait, replacing it with special archive type
- PR #1556[3353] - Executor algorithms
- PR #1555[3354] - Remove the necessity to specify archive flags on the receiving end
- PR #1554[3355] - Removing obsolete Boost.Serialization macros
- PR #1553[3356] - Properly fix HPX_DEFINE_*_ACTION macros
- PR #1552[3357] - Fixed algorithms relying on copy_if implementation
- PR #1551[3358] - Pxfs - Modifying FindOrangeFS.cmake based on OrangeFS 2.9.X
- Issue #1550[3359] - Passing plain identifier inside HPX_DEFINE_PLAIN_ACTION_1
- PR #1549[3360] - Fixing intel14/libstdc++4.4
- PR #1548[3361] - Moving raw_ptr to detail namespace
- PR #1547[3362] - Adding support for executors to future.then
- PR #1546[3363] - Executor traits result types
- PR #1545[3364] - Integrate executors with dataflow
- PR #1543[3365] - Fix potential zero-copy for primarynamespace::bulk_service_async et.al.
- PR #1542[3366] - Merging HPX0.9.10 into pxfs branch

[3344] https://github.com/STEllAR-GROUP/hpx/issues/1566
[3345] https://github.com/STEllAR-GROUP/hpx/pull/1565
[3346] https://github.com/STEllAR-GROUP/hpx/pull/1564
[3347] https://github.com/STEllAR-GROUP/hpx/pull/1563
[3348] https://github.com/STEllAR-GROUP/hpx/pull/1562
[3349] https://github.com/STEllAR-GROUP/hpx/pull/1561
[3350] https://github.com/STEllAR-GROUP/hpx/issues/1560
[3351] https://github.com/STEllAR-GROUP/hpx/pull/1558
[3352] https://github.com/STEllAR-GROUP/hpx/pull/1557
[3353] https://github.com/STEllAR-GROUP/hpx/pull/1556
[3354] https://github.com/STEllAR-GROUP/hpx/pull/1555
[3355] https://github.com/STEllAR-GROUP/hpx/pull/1554
[3356] https://github.com/STEllAR-GROUP/hpx/pull/1553
[3357] https://github.com/STEllAR-GROUP/hpx/pull/1552
[3358] https://github.com/STEllAR-GROUP/hpx/pull/1551
[3359] https://github.com/STEllAR-GROUP/hpx/issues/1550
[3360] https://github.com/STEllAR-GROUP/hpx/pull/1549
[3361] https://github.com/STEllAR-GROUP/hpx/pull/1548
[3362] https://github.com/STEllAR-GROUP/hpx/pull/1547
[3363] https://github.com/STEllAR-GROUP/hpx/pull/1546
[3364] https://github.com/STEllAR-GROUP/hpx/pull/1545
[3365] https://github.com/STEllAR-GROUP/hpx/pull/1543
[3366] https://github.com/STEllAR-GROUP/hpx/pull/1542

- PR #1541[3367] - Removed stale cmake tests, unused since the great cmake refactoring
- PR #1540[3368] - Fix idle-rate on platforms without TSC
- PR #1539[3369] - Reporting situation if zero-copy-serialization was performed by a parcel generated from a plain apply/async
- PR #1538[3370] - Changed return type of bulk executors and added test
- Issue #1537[3371] - Incorrect cpuid config tests
- PR #1536[3372] - Changed return type of bulk executors and added test
- PR #1535[3373] - Make sure promise::get_gid() can be called more than once
- PR #1534[3374] - Fixed async_callback with bound callback
- PR #1533[3375] - Updated the link in the documentation to a publically- accessible URL
- PR #1532[3376] - Make sure sync primitives are not copyable nor movable
- PR #1531[3377] - Fix unwrapped issue with future ranges of void type
- PR #1530[3378] - Serialization complex
- Issue #1528[3379] - Unwrapped issue with future<void>
- Issue #1527[3380] - HPX does not build with Boost 1.58.0
- PR #1526[3381] - Added support for boost.multi_array serialization
- PR #1525[3382] - Properly handle deferred futures, fixes #1506
- PR #1524[3383] - Making sure invalid action argument types generate clear error message
- Issue #1522[3384] - Need serialization support for boost multi array
- Issue #1521[3385] - Remote async and zero-copy serialization optimizations don't play well together
- PR #1520[3386] - Fixing UB whil registering polymorphic classes for serialization
- PR #1519[3387] - Making detail::condition_variable safe to use
- PR #1518[3388] - Fix when_some bug missing indices in its result
- Issue #1517[3389] - Typo may affect CMake build system tests

3367 https://github.com/STEllAR-GROUP/hpx/pull/1541
3368 https://github.com/STEllAR-GROUP/hpx/pull/1540
3369 https://github.com/STEllAR-GROUP/hpx/pull/1539
3370 https://github.com/STEllAR-GROUP/hpx/pull/1538
3371 https://github.com/STEllAR-GROUP/hpx/issues/1537
3372 https://github.com/STEllAR-GROUP/hpx/pull/1536
3373 https://github.com/STEllAR-GROUP/hpx/pull/1535
3374 https://github.com/STEllAR-GROUP/hpx/pull/1534
3375 https://github.com/STEllAR-GROUP/hpx/pull/1533
3376 https://github.com/STEllAR-GROUP/hpx/pull/1532
3377 https://github.com/STEllAR-GROUP/hpx/pull/1531
3378 https://github.com/STEllAR-GROUP/hpx/pull/1530
3379 https://github.com/STEllAR-GROUP/hpx/issues/1528
3380 https://github.com/STEllAR-GROUP/hpx/issues/1527
3381 https://github.com/STEllAR-GROUP/hpx/pull/1526
3382 https://github.com/STEllAR-GROUP/hpx/pull/1525
3383 https://github.com/STEllAR-GROUP/hpx/pull/1524
3384 https://github.com/STEllAR-GROUP/hpx/issues/1522
3385 https://github.com/STEllAR-GROUP/hpx/issues/1521
3386 https://github.com/STEllAR-GROUP/hpx/pull/1520
3387 https://github.com/STEllAR-GROUP/hpx/pull/1519
3388 https://github.com/STEllAR-GROUP/hpx/pull/1518
3389 https://github.com/STEllAR-GROUP/hpx/issues/1517

- PR #1516[3390] - Fixing Posix context
- PR #1515[3391] - Fixing Posix context
- PR #1514[3392] - Correct problems with loading dynamic components
- PR #1513[3393] - Fixing intel glibc4 4
- Issue #1508[3394] - memory and papi counters do not work
- Issue #1507[3395] - Unrecognized Command Line Option Error causing exit status 0
- Issue #1506[3396] - Properly handle deferred futures
- PR #1505[3397] - Adding #include - would not compile without this
- Issue #1502[3398] - `boost::filesystem::exists` throws unexpected exception
- Issue #1501[3399] - hwloc configuration options are wrong for MIC
- PR #1504[3400] - Making sure boost::filesystem::exists() does not throw
- PR #1500[3401] - Exit application on `--hpx:version/-v` and `--hpx:info`
- PR #1498[3402] - Extended task block
- PR #1497[3403] - Unique ptr serialization
- PR #1496[3404] - Unique ptr serialization (closed)
- PR #1495[3405] - Switching circleci build type to debug
- Issue #1494[3406] - `--hpx:version/-v` does not exit after printing version information
- Issue #1493[3407] - add an `hpx_` prefix to libraries and components to avoid name conflicts
- Issue #1492[3408] - Define and ensure limitations for arguments to async/apply
- PR #1489[3409] - Enable idle rate counter on demand
- PR #1488[3410] - Made sure `detail::condition_variable` can be safely destroyed
- PR #1487[3411] - Introduced default (main) template implementation for `ignore_while_checking`
- PR #1486[3412] - Add HPX inspect tool

[3390] https://github.com/STEllAR-GROUP/hpx/pull/1516
[3391] https://github.com/STEllAR-GROUP/hpx/pull/1515
[3392] https://github.com/STEllAR-GROUP/hpx/pull/1514
[3393] https://github.com/STEllAR-GROUP/hpx/pull/1513
[3394] https://github.com/STEllAR-GROUP/hpx/issues/1508
[3395] https://github.com/STEllAR-GROUP/hpx/issues/1507
[3396] https://github.com/STEllAR-GROUP/hpx/issues/1506
[3397] https://github.com/STEllAR-GROUP/hpx/pull/1505
[3398] https://github.com/STEllAR-GROUP/hpx/issues/1502
[3399] https://github.com/STEllAR-GROUP/hpx/issues/1501
[3400] https://github.com/STEllAR-GROUP/hpx/pull/1504
[3401] https://github.com/STEllAR-GROUP/hpx/pull/1500
[3402] https://github.com/STEllAR-GROUP/hpx/pull/1498
[3403] https://github.com/STEllAR-GROUP/hpx/pull/1497
[3404] https://github.com/STEllAR-GROUP/hpx/pull/1496
[3405] https://github.com/STEllAR-GROUP/hpx/pull/1495
[3406] https://github.com/STEllAR-GROUP/hpx/issues/1494
[3407] https://github.com/STEllAR-GROUP/hpx/issues/1493
[3408] https://github.com/STEllAR-GROUP/hpx/issues/1492
[3409] https://github.com/STEllAR-GROUP/hpx/pull/1489
[3410] https://github.com/STEllAR-GROUP/hpx/pull/1488
[3411] https://github.com/STEllAR-GROUP/hpx/pull/1487
[3412] https://github.com/STEllAR-GROUP/hpx/pull/1486

- Issue #1485[3413] - `ignore_while_locked` doesn't support all Lockable types
- PR #1484[3414] - Docker image generation
- PR #1483[3415] - Move external endian library into HPX
- PR #1482[3416] - Actions with integer type ids
- Issue #1481[3417] - Sync primitives safe destruction
- Issue #1480[3418] - Move external/boost/endian into hpx/util
- Issue #1478[3419] - Boost inspect violations
- PR #1479[3420] - Adds serialization for arrays; some further/minor fixes
- PR #1477[3421] - Fixing problems with the Intel compiler using a GCC 4.4 std library
- PR #1476[3422] - Adding `hpx::lcos::latch` and `hpx::lcos::local::latch`
- Issue #1475[3423] - Boost inspect violations
- PR #1473[3424] - Fixing action move tests
- Issue #1471[3425] - Sync primitives should not be movable
- PR #1470[3426] - Removing `hpx::util::polymorphic_factory`
- PR #1468[3427] - Fixed container creation
- Issue #1467[3428] - HPX application fail during finalization
- Issue #1466[3429] - HPX doesn't pick up Torque's nodefile on SuperMIC
- Issue #1464[3430] - HPX option for pre and post bootstrap performance counters
- PR #1463[3431] - Replacing `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1462[3432] - Consolidated task_region with N4411
- PR #1461[3433] - Consolidate inconsistent CMake option names
- Issue #1460[3434] - Which malloc is actually used? or at least which one is HPX built with
- Issue #1459[3435] - Make cmake configure step fail explicitly if compiler version is not supported

3413 https://github.com/STEllAR-GROUP/hpx/issues/1485
3414 https://github.com/STEllAR-GROUP/hpx/pull/1484
3415 https://github.com/STEllAR-GROUP/hpx/pull/1483
3416 https://github.com/STEllAR-GROUP/hpx/pull/1482
3417 https://github.com/STEllAR-GROUP/hpx/issues/1481
3418 https://github.com/STEllAR-GROUP/hpx/issues/1480
3419 https://github.com/STEllAR-GROUP/hpx/issues/1478
3420 https://github.com/STEllAR-GROUP/hpx/pull/1479
3421 https://github.com/STEllAR-GROUP/hpx/pull/1477
3422 https://github.com/STEllAR-GROUP/hpx/pull/1476
3423 https://github.com/STEllAR-GROUP/hpx/issues/1475
3424 https://github.com/STEllAR-GROUP/hpx/pull/1473
3425 https://github.com/STEllAR-GROUP/hpx/issues/1471
3426 https://github.com/STEllAR-GROUP/hpx/pull/1470
3427 https://github.com/STEllAR-GROUP/hpx/pull/1468
3428 https://github.com/STEllAR-GROUP/hpx/issues/1467
3429 https://github.com/STEllAR-GROUP/hpx/issues/1466
3430 https://github.com/STEllAR-GROUP/hpx/issues/1464
3431 https://github.com/STEllAR-GROUP/hpx/pull/1463
3432 https://github.com/STEllAR-GROUP/hpx/pull/1462
3433 https://github.com/STEllAR-GROUP/hpx/pull/1461
3434 https://github.com/STEllAR-GROUP/hpx/issues/1460
3435 https://github.com/STEllAR-GROUP/hpx/issues/1459

- Issue #1458[3436] - Update `parallel::task_region` with N4411
- PR #1456[3437] - Consolidating `new_<>()`
- Issue #1455[3438] - Replace `async_colocated(id, ...)` with `async(colocated(id), ...)`
- PR #1454[3439] - Removed harmful std::moves from return statements
- PR #1453[3440] - Use range-based for-loop instead of Boost.Foreach
- PR #1452[3441] - C++ feature tests
- PR #1451[3442] - When serializing, pass archive flags to traits::get_type_size
- Issue #1450[3443] - traits:get_type_size needs archive flags to enable zero_copy optimizations
- Issue #1449[3444] - "couldn't create performance counter" - AGAS
- Issue #1448[3445] - Replace distributing factories with `new_<T[]>(...)`
- PR #1447[3446] - Removing obsolete remote_object component
- PR #1446[3447] - Hpx serialization
- PR #1445[3448] - Replacing travis with circleci
- PR #1443[3449] - Always stripping HPX command line arguments before executing start function
- PR #1442[3450] - Adding –hpx:bind=none to disable thread affinities
- Issue #1439[3451] - Libraries get linked in multiple times, RPATH is not properly set
- PR #1438[3452] - Removed superfluous typedefs
- Issue #1437[3453] - `hpx::init()` should strip HPX-related flags from argv
- Issue #1436[3454] - Add strong scaling option to htts
- PR #1435[3455] - Adding `async_cb`, `async_continue_cb`, and `async_colocated_cb`
- PR #1434[3456] - Added missing install rule, removed some dead CMake code
- PR #1433[3457] - Add GitExternal and SubProject cmake scripts from eyescale/cmake repo
- Issue #1432[3458] - Add command line flag to disable thread pinning

3436 https://github.com/STEllAR-GROUP/hpx/issues/1458
3437 https://github.com/STEllAR-GROUP/hpx/pull/1456
3438 https://github.com/STEllAR-GROUP/hpx/issues/1455
3439 https://github.com/STEllAR-GROUP/hpx/pull/1454
3440 https://github.com/STEllAR-GROUP/hpx/pull/1453
3441 https://github.com/STEllAR-GROUP/hpx/pull/1452
3442 https://github.com/STEllAR-GROUP/hpx/pull/1451
3443 https://github.com/STEllAR-GROUP/hpx/issues/1450
3444 https://github.com/STEllAR-GROUP/hpx/issues/1449
3445 https://github.com/STEllAR-GROUP/hpx/issues/1448
3446 https://github.com/STEllAR-GROUP/hpx/pull/1447
3447 https://github.com/STEllAR-GROUP/hpx/pull/1446
3448 https://github.com/STEllAR-GROUP/hpx/pull/1445
3449 https://github.com/STEllAR-GROUP/hpx/pull/1443
3450 https://github.com/STEllAR-GROUP/hpx/pull/1442
3451 https://github.com/STEllAR-GROUP/hpx/issues/1439
3452 https://github.com/STEllAR-GROUP/hpx/pull/1438
3453 https://github.com/STEllAR-GROUP/hpx/issues/1437
3454 https://github.com/STEllAR-GROUP/hpx/issues/1436
3455 https://github.com/STEllAR-GROUP/hpx/pull/1435
3456 https://github.com/STEllAR-GROUP/hpx/pull/1434
3457 https://github.com/STEllAR-GROUP/hpx/pull/1433
3458 https://github.com/STEllAR-GROUP/hpx/issues/1432

- PR #1431[3459] - Fix #1423
- Issue #1430[3460] - Inconsistent CMake option names
- Issue #1429[3461] - Configure setting `HPX_HAVE_PARCELPORT_MPI` is ignored
- PR #1428[3462] - Fixes #1419 (closed)
- PR #1427[3463] - Adding stencil_iterator and transform_iterator
- PR #1426[3464] - Fixes #1419
- PR #1425[3465] - During serialization memory allocation should honour allocator chunk size
- Issue #1424[3466] - chunk allocation during serialization does not use memory pool/allocator chunk size
- Issue #1423[3467] - Remove `HPX_STD_UNIQUE_PTR`
- Issue #1422[3468] - hpx:threads=all allocates too many os threads
- PR #1420[3469] - added .travis.yml
- Issue #1419[3470] - Unify enums: `hpx::runtime::state` and `hpx::state`
- PR #1416[3471] - Adding travis builder
- Issue #1414[3472] - Correct directory for dispatch_gcc46.hpp iteration
- Issue #1410[3473] - Set operation algorithms
- Issue #1389[3474] - Parallel algorithms relying on scan partitioner break for small number of elements
- Issue #1325[3475] - Exceptions thrown during parcel handling are not handled correctly
- Issue #1315[3476] - Errors while running performance tests
- Issue #1309[3477] - `hpx::vector` partitions are not easily extendable by applications
- PR #1300[3478] - Added serialization/de-serialization to examples.tuplespace
- Issue #1251[3479] - hpx::threads::get_thread_count doesn't consider pending threads
- Issue #1008[3480] - Decrease in application performance overtime; occasional spikes of major slowdown
- Issue #1001[3481] - Zero copy serialization raises assert

[3459] https://github.com/STEllAR-GROUP/hpx/pull/1431
[3460] https://github.com/STEllAR-GROUP/hpx/issues/1430
[3461] https://github.com/STEllAR-GROUP/hpx/issues/1429
[3462] https://github.com/STEllAR-GROUP/hpx/pull/1428
[3463] https://github.com/STEllAR-GROUP/hpx/pull/1427
[3464] https://github.com/STEllAR-GROUP/hpx/pull/1426
[3465] https://github.com/STEllAR-GROUP/hpx/pull/1425
[3466] https://github.com/STEllAR-GROUP/hpx/issues/1424
[3467] https://github.com/STEllAR-GROUP/hpx/issues/1423
[3468] https://github.com/STEllAR-GROUP/hpx/issues/1422
[3469] https://github.com/STEllAR-GROUP/hpx/pull/1420
[3470] https://github.com/STEllAR-GROUP/hpx/issues/1419
[3471] https://github.com/STEllAR-GROUP/hpx/pull/1416
[3472] https://github.com/STEllAR-GROUP/hpx/issues/1414
[3473] https://github.com/STEllAR-GROUP/hpx/issues/1410
[3474] https://github.com/STEllAR-GROUP/hpx/issues/1389
[3475] https://github.com/STEllAR-GROUP/hpx/issues/1325
[3476] https://github.com/STEllAR-GROUP/hpx/issues/1315
[3477] https://github.com/STEllAR-GROUP/hpx/issues/1309
[3478] https://github.com/STEllAR-GROUP/hpx/pull/1300
[3479] https://github.com/STEllAR-GROUP/hpx/issues/1251
[3480] https://github.com/STEllAR-GROUP/hpx/issues/1008
[3481] https://github.com/STEllAR-GROUP/hpx/issues/1001

- Issue #721[3482] - Make HPX usable for Xeon Phi
- Issue #524[3483] - Extend scheduler to support threads which can't be stolen

2.11.12 *HPX* V0.9.10 (Mar 24, 2015)

General changes

This is the 12th official release of *HPX*. It coincides with the 7th anniversary of the first commit to our source code repository. Since then, we have seen over 12300 commits amounting to more than 220000 lines of C++ code.

The major focus of this release was to improve the reliability of large scale runs. We believe to have achieved this goal as we now can reliably run *HPX* applications on up to ~24k cores. We have also shown that HPX can be used with success for symmetric runs (applications using both, host cores and Intel Xeon/Phi coprocessors). This is a huge step forward in terms of the usability of *HPX*. The main focus of this work involved isolating the causes of the segmentation faults at start up and shut down. Many of these issues were discovered to be the result of the suspension of threads which hold locks.

A very important improvement introduced with this release is the refactoring of the code representing our parcel-port implementation. Parcel- ports can now be implemented by 3rd parties as independent plugins which are dynamically loaded at runtime (static linking of parcel-ports is also supported). This refactoring also includes a massive improvement of the performance of our existing parcel-ports. We were able to significantly reduce the networking latencies and to improve the available networking bandwidth. Please note that in this release we disabled the ibverbs and ipc parcel ports as those have not been ported to the new plugin system yet (see Issue #839[3484]).

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[3485] 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

[3482] https://github.com/STEllAR-GROUP/hpx/issues/721
[3483] https://github.com/STEllAR-GROUP/hpx/issues/524
[3484] https://github.com/STEllAR-GROUP/hpx/issues/839
[3485] https://github.com/STEllAR-GROUP/hpx/issues/1338

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[3486] - Internal shared_future serialization copies
- Issue #1399[3487] - Build takes unusually long time…
- Issue #1398[3488] - Tests using the scan partitioner are broken on at least gcc 4.7 and intel compiler
- Issue #1397[3489] - Completely remove hpx::unique_future
- Issue #1396[3490] - Parallel scan algorithms with different initial values
- Issue #1395[3491] - Race Condition - 1d_stencil_8 - SuperMIC
- Issue #1394[3492] - "suspending thread while at least one lock is being held" - 1d_stencil_8 - SuperMIC
- Issue #1393[3493] - SEGFAULT in 1d_stencil_8 on SuperMIC
- Issue #1392[3494] - Fixing #1168
- Issue #1391[3495] - Parallel Algorithms for scan partitioner for small number of elements
- Issue #1387[3496] - Failure with more than 4 localities
- Issue #1386[3497] - Dispatching unhandled exceptions to outer user code
- Issue #1385[3498] - Adding Copy algorithms, fixing parallel::copy_if
- Issue #1384[3499] - Fixing 1325
- Issue #1383[3500] - Fixed #504: Refactor Dataflow LCO to work with futures, this removes the dataflow component as it is obsolete
- Issue #1382[3501] - is_sorted, is_sorted_until and is_partitioned algorithms
- Issue #1381[3502] - fix for CMake versions prior to 3.1

3486 https://github.com/STEllAR-GROUP/hpx/issues/1402
3487 https://github.com/STEllAR-GROUP/hpx/issues/1399
3488 https://github.com/STEllAR-GROUP/hpx/issues/1398
3489 https://github.com/STEllAR-GROUP/hpx/issues/1397
3490 https://github.com/STEllAR-GROUP/hpx/issues/1396
3491 https://github.com/STEllAR-GROUP/hpx/issues/1395
3492 https://github.com/STEllAR-GROUP/hpx/issues/1394
3493 https://github.com/STEllAR-GROUP/hpx/issues/1393
3494 https://github.com/STEllAR-GROUP/hpx/issues/1392
3495 https://github.com/STEllAR-GROUP/hpx/issues/1391
3496 https://github.com/STEllAR-GROUP/hpx/issues/1387
3497 https://github.com/STEllAR-GROUP/hpx/issues/1386
3498 https://github.com/STEllAR-GROUP/hpx/issues/1385
3499 https://github.com/STEllAR-GROUP/hpx/issues/1384
3500 https://github.com/STEllAR-GROUP/hpx/issues/1383
3501 https://github.com/STEllAR-GROUP/hpx/issues/1382
3502 https://github.com/STEllAR-GROUP/hpx/issues/1381

- Issue #1380[3503] - resolved warning in CMake 3.1 and newer
- Issue #1379[3504] - Compilation error with papi
- Issue #1378[3505] - Towards safer migration
- Issue #1377[3506] - HPXConfig.cmake should include `TCMALLOC_LIBRARY` and `TCMALLOC_INCLUDE_DIR`
- Issue #1376[3507] - Warning on uninitialized member
- Issue #1375[3508] - Fixing 1163
- Issue #1374[3509] - Fixing the MSVC 12 release builder
- Issue #1373[3510] - Modifying parallel search algorithm for zero length searches
- Issue #1372[3511] - Modifying parallel search algorithm for zero length searches
- Issue #1371[3512] - Avoid holding a lock during agas::incref while doing a credit split
- Issue #1370[3513] - `--hpx:bind` throws unexpected error
- Issue #1369[3514] - Getting rid of (void) in loops
- Issue #1368[3515] - Variadic templates support for tuple
- Issue #1367[3516] - One last batch of variadic templates support
- Issue #1366[3517] - Fixing symbolic namespace hang
- Issue #1365[3518] - More held locks
- Issue #1364[3519] - Add counters 1363
- Issue #1363[3520] - Add thread overhead counters
- Issue #1362[3521] - Std config removal
- Issue #1361[3522] - Parcelport plugins
- Issue #1360[3523] - Detuplify transfer_action
- Issue #1359[3524] - Removed obsolete checks
- Issue #1358[3525] - Fixing 1352

3503 https://github.com/STEllAR-GROUP/hpx/issues/1380
3504 https://github.com/STEllAR-GROUP/hpx/issues/1379
3505 https://github.com/STEllAR-GROUP/hpx/issues/1378
3506 https://github.com/STEllAR-GROUP/hpx/issues/1377
3507 https://github.com/STEllAR-GROUP/hpx/issues/1376
3508 https://github.com/STEllAR-GROUP/hpx/issues/1375
3509 https://github.com/STEllAR-GROUP/hpx/issues/1374
3510 https://github.com/STEllAR-GROUP/hpx/issues/1373
3511 https://github.com/STEllAR-GROUP/hpx/issues/1372
3512 https://github.com/STEllAR-GROUP/hpx/issues/1371
3513 https://github.com/STEllAR-GROUP/hpx/issues/1370
3514 https://github.com/STEllAR-GROUP/hpx/issues/1369
3515 https://github.com/STEllAR-GROUP/hpx/issues/1368
3516 https://github.com/STEllAR-GROUP/hpx/issues/1367
3517 https://github.com/STEllAR-GROUP/hpx/issues/1366
3518 https://github.com/STEllAR-GROUP/hpx/issues/1365
3519 https://github.com/STEllAR-GROUP/hpx/issues/1364
3520 https://github.com/STEllAR-GROUP/hpx/issues/1363
3521 https://github.com/STEllAR-GROUP/hpx/issues/1362
3522 https://github.com/STEllAR-GROUP/hpx/issues/1361
3523 https://github.com/STEllAR-GROUP/hpx/issues/1360
3524 https://github.com/STEllAR-GROUP/hpx/issues/1359
3525 https://github.com/STEllAR-GROUP/hpx/issues/1358

- Issue #1357[3526] - Variadic templates support for runtime_support and components
- Issue #1356[3527] - fixed coordinate test for intel13
- Issue #1355[3528] - fixed coordinate.hpp
- Issue #1354[3529] - Lexicographical Compare completed
- Issue #1353[3530] - HPX should set `Boost_ADDITIONAL_VERSIONS` flags
- Issue #1352[3531] - Error: Cannot find action '' in type registry: HPX(bad_action_code)
- Issue #1351[3532] - Variadic templates support for appliers
- Issue #1350[3533] - Actions simplification
- Issue #1349[3534] - Variadic when and wait functions
- Issue #1348[3535] - Added hpx_init header to test files
- Issue #1347[3536] - Another batch of variadic templates support
- Issue #1346[3537] - Segmented copy
- Issue #1345[3538] - Attempting to fix hangs during shutdown
- Issue #1344[3539] - Std config removal
- Issue #1343[3540] - Removing various distribution policies for hpx::vector
- Issue #1342[3541] - Inclusive scan
- Issue #1341[3542] - Exclusive scan
- Issue #1340[3543] - Adding `parallel::count` for distributed data structures, adding tests
- Issue #1339[3544] - Update argument order for transform_reduce
- Issue #1337[3545] - Fix dataflow to handle properly ranges of futures
- Issue #1336[3546] - dataflow needs to hold onto futures passed to it
- Issue #1335[3547] - Fails to compile with msvc14
- Issue #1334[3548] - Examples build problem

3526 https://github.com/STEllAR-GROUP/hpx/issues/1357
3527 https://github.com/STEllAR-GROUP/hpx/issues/1356
3528 https://github.com/STEllAR-GROUP/hpx/issues/1355
3529 https://github.com/STEllAR-GROUP/hpx/issues/1354
3530 https://github.com/STEllAR-GROUP/hpx/issues/1353
3531 https://github.com/STEllAR-GROUP/hpx/issues/1352
3532 https://github.com/STEllAR-GROUP/hpx/issues/1351
3533 https://github.com/STEllAR-GROUP/hpx/issues/1350
3534 https://github.com/STEllAR-GROUP/hpx/issues/1349
3535 https://github.com/STEllAR-GROUP/hpx/issues/1348
3536 https://github.com/STEllAR-GROUP/hpx/issues/1347
3537 https://github.com/STEllAR-GROUP/hpx/issues/1346
3538 https://github.com/STEllAR-GROUP/hpx/issues/1345
3539 https://github.com/STEllAR-GROUP/hpx/issues/1344
3540 https://github.com/STEllAR-GROUP/hpx/issues/1343
3541 https://github.com/STEllAR-GROUP/hpx/issues/1342
3542 https://github.com/STEllAR-GROUP/hpx/issues/1341
3543 https://github.com/STEllAR-GROUP/hpx/issues/1340
3544 https://github.com/STEllAR-GROUP/hpx/issues/1339
3545 https://github.com/STEllAR-GROUP/hpx/issues/1337
3546 https://github.com/STEllAR-GROUP/hpx/issues/1336
3547 https://github.com/STEllAR-GROUP/hpx/issues/1335
3548 https://github.com/STEllAR-GROUP/hpx/issues/1334

- Issue #1333[3549] - Distributed transform reduce
- Issue #1332[3550] - Variadic templates support for actions
- Issue #1331[3551] - Some ambiguous calls of map::erase have been prevented by adding additional check in locality constructor.
- Issue #1330[3552] - Defining Plain Actions does not work as described in the documentation
- Issue #1329[3553] - Distributed vector cleanup
- Issue #1328[3554] - Sync docs and comments with code in hello_world example
- Issue #1327[3555] - Typos in docs
- Issue #1326[3556] - Documentation and code diverged in Fibonacci tutorial
- Issue #1325[3557] - Exceptions thrown during parcel handling are not handled correctly
- Issue #1324[3558] - fixed bandwidth calculation
- Issue #1323[3559] - mmap() failed to allocate thread stack due to insufficient resources
- Issue #1322[3560] - HPX fails to build aa182cf
- Issue #1321[3561] - Limiting size of outgoing messages while coalescing parcels
- Issue #1320[3562] - passing a future with launch::deferred in remote function call causes hang
- Issue #1319[3563] - An exception when tries to specify number high priority threads with abp-priority
- Issue #1318[3564] - Unable to run program with abp-priority and numa-sensitivity enabled
- Issue #1317[3565] - N4071 Search/Search_n finished, minor changes
- Issue #1316[3566] - Add config option to make -Ihpx.run_hpx_main!=1 the default
- Issue #1314[3567] - Variadic support for async and apply
- Issue #1313[3568] - Adjust when_any/some to the latest proposed interfaces
- Issue #1312[3569] - Fixing #857: hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #1311[3570] - Distributed get'er/set'er_values for distributed vector

[3549] https://github.com/STEllAR-GROUP/hpx/issues/1333
[3550] https://github.com/STEllAR-GROUP/hpx/issues/1332
[3551] https://github.com/STEllAR-GROUP/hpx/issues/1331
[3552] https://github.com/STEllAR-GROUP/hpx/issues/1330
[3553] https://github.com/STEllAR-GROUP/hpx/issues/1329
[3554] https://github.com/STEllAR-GROUP/hpx/issues/1328
[3555] https://github.com/STEllAR-GROUP/hpx/issues/1327
[3556] https://github.com/STEllAR-GROUP/hpx/issues/1326
[3557] https://github.com/STEllAR-GROUP/hpx/issues/1325
[3558] https://github.com/STEllAR-GROUP/hpx/issues/1324
[3559] https://github.com/STEllAR-GROUP/hpx/issues/1323
[3560] https://github.com/STEllAR-GROUP/hpx/issues/1322
[3561] https://github.com/STEllAR-GROUP/hpx/issues/1321
[3562] https://github.com/STEllAR-GROUP/hpx/issues/1320
[3563] https://github.com/STEllAR-GROUP/hpx/issues/1319
[3564] https://github.com/STEllAR-GROUP/hpx/issues/1318
[3565] https://github.com/STEllAR-GROUP/hpx/issues/1317
[3566] https://github.com/STEllAR-GROUP/hpx/issues/1316
[3567] https://github.com/STEllAR-GROUP/hpx/issues/1314
[3568] https://github.com/STEllAR-GROUP/hpx/issues/1313
[3569] https://github.com/STEllAR-GROUP/hpx/issues/1312
[3570] https://github.com/STEllAR-GROUP/hpx/issues/1311

- Issue #1310[3571] - Crashing in hpx::parcelset::policies::mpi::connection_handler::handle_messages() on SuperMIC
- Issue #1308[3572] - Unable to execute an application with –hpx:threads
- Issue #1307[3573] - merge_graph linking issue
- Issue #1306[3574] - First batch of variadic templates support
- Issue #1305[3575] - Create a compiler wrapper
- Issue #1304[3576] - Provide a compiler wrapper for hpx
- Issue #1303[3577] - Drop support for GCC44
- Issue #1302[3578] - Fixing #1297
- Issue #1301[3579] - Compilation error when tried to use boost range iterators with wait_all
- Issue #1298[3580] - Distributed vector
- Issue #1297[3581] - Unable to invoke component actions recursively
- Issue #1294[3582] - HDF5 build error
- Issue #1275[3583] - The parcelport implementation is non-optimal
- Issue #1267[3584] - Added classes and unit tests for local_file, orangefs_file and pxfs_file
- Issue #1264[3585] - Error "assertion '!m_fun' failed" randomly occurs when using TCP
- Issue #1254[3586] - thread binding seems to not work properly
- Issue #1220[3587] - parallel::copy_if is broken
- Issue #1217[3588] - Find a better way of fixing the issue patched by #1216
- Issue #1168[3589] - Starting HPX on Cray machines using aprun isn't working correctly
- Issue #1085[3590] - Replace startup and shutdown barriers with broadcasts
- Issue #981[3591] - With SLURM, –hpx:threads=8 should not be necessary
- Issue #857[3592] - hpx::naming::locality leaks parcelport specific information into the public interface
- Issue #850[3593] - "flush" not documented

3571 https://github.com/STEllAR-GROUP/hpx/issues/1310
3572 https://github.com/STEllAR-GROUP/hpx/issues/1308
3573 https://github.com/STEllAR-GROUP/hpx/issues/1307
3574 https://github.com/STEllAR-GROUP/hpx/issues/1306
3575 https://github.com/STEllAR-GROUP/hpx/issues/1305
3576 https://github.com/STEllAR-GROUP/hpx/issues/1304
3577 https://github.com/STEllAR-GROUP/hpx/issues/1303
3578 https://github.com/STEllAR-GROUP/hpx/issues/1302
3579 https://github.com/STEllAR-GROUP/hpx/issues/1301
3580 https://github.com/STEllAR-GROUP/hpx/issues/1298
3581 https://github.com/STEllAR-GROUP/hpx/issues/1297
3582 https://github.com/STEllAR-GROUP/hpx/issues/1294
3583 https://github.com/STEllAR-GROUP/hpx/issues/1275
3584 https://github.com/STEllAR-GROUP/hpx/issues/1267
3585 https://github.com/STEllAR-GROUP/hpx/issues/1264
3586 https://github.com/STEllAR-GROUP/hpx/issues/1254
3587 https://github.com/STEllAR-GROUP/hpx/issues/1220
3588 https://github.com/STEllAR-GROUP/hpx/issues/1217
3589 https://github.com/STEllAR-GROUP/hpx/issues/1168
3590 https://github.com/STEllAR-GROUP/hpx/issues/1085
3591 https://github.com/STEllAR-GROUP/hpx/issues/981
3592 https://github.com/STEllAR-GROUP/hpx/issues/857
3593 https://github.com/STEllAR-GROUP/hpx/issues/850

- Issue #763[3594] - Create buildbot instance that uses std::bind as HPX_STD_BIND
- Issue #680[3595] - Convert parcel ports into a plugin system
- Issue #582[3596] - Make exception thrown from HPX threads available from `hpx::init`
- Issue #504[3597] - Refactor Dataflow LCO to work with futures
- Issue #196[3598] - Don't store copies of the locality network metadata in the gva table

2.11.13 *HPX* V0.9.9 (Oct 31, 2014, codename Spooky)

General changes

We have had over 1500 commits since the last release and we have closed over 200 tickets (bugs, feature requests, pull requests, etc.). These are by far the largest numbers of commits and resolved issues for any of the *HPX* releases so far. We are especially happy about the large number of people who contributed for the first time to *HPX*.

- We completed the transition from the older (non-conforming) implementation of `hpx::future` to the new and fully conforming version by removing the old code and by renaming the type `hpx::unique_future` to `hpx::future`. In order to maintain backwards compatibility with existing code which uses the type `hpx::unique_future` we support the configuration variable `HPX_UNIQUE_FUTURE_ALIAS`. If this variable is set to `ON` while running cmake it will additionally define a template alias for this type.
- We rewrote and significantly changed our build system. Please have a look at the new (now generated) documentation here: *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[3599].
- We started to implement various proposals to the C++ Standardization committee related to parallelism and concurrency, most notably N4409[3600] (Working Draft, Technical Specification for C++ Extensions for Parallelism), N4411[3601] (Task Region Rev. 3), and N4313[3602] (Working Draft, Technical Specification for C++ Extensions for Concurrency).

[3594] https://github.com/STEllAR-GROUP/hpx/issues/763
[3595] https://github.com/STEllAR-GROUP/hpx/issues/680
[3596] https://github.com/STEllAR-GROUP/hpx/issues/582
[3597] https://github.com/STEllAR-GROUP/hpx/issues/504
[3598] https://github.com/STEllAR-GROUP/hpx/issues/196
[3599] https://github.com/STEllAR-GROUP/hpx/issues/1204
[3600] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4409.pdf
[3601] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
[3602] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html

- 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[3603].

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1296[3604] - Rename make_error_future to make_exceptional_future, adjust to N4123
- Issue #1295[3605] - building issue
- Issue #1293[3606] - Transpose example
- Issue #1292[3607] - Wrong abs() function used in example
- Issue #1291[3608] - non-synchronized shift operators have been removed
- Issue #1290[3609] - RDTSCP is defined as true for Xeon Phi build
- Issue #1289[3610] - Fixing 1288
- Issue #1288[3611] - Add new performance counters
- Issue #1287[3612] - Hierarchy scheduler broken performance counters
- Issue #1286[3613] - Algorithm cleanup
- Issue #1285[3614] - Broken Links in Documentation
- Issue #1284[3615] - Uninitialized copy
- Issue #1283[3616] - missing boost::scoped_ptr includes
- Issue #1282[3617] - Update documentation of build options for schedulers
- Issue #1281[3618] - reset idle rate counter
- Issue #1280[3619] - Bug when executing on Intel MIC
- Issue #1279[3620] - Add improved when_all/wait_all
- Issue #1278[3621] - Implement improved when_all/wait_all
- Issue #1277[3622] - feature request: get access to argc argv and variables_map

[3603] http://rostam.cct.lsu.edu/
[3604] https://github.com/STEllAR-GROUP/hpx/issues/1296
[3605] https://github.com/STEllAR-GROUP/hpx/issues/1295
[3606] https://github.com/STEllAR-GROUP/hpx/issues/1293
[3607] https://github.com/STEllAR-GROUP/hpx/issues/1292
[3608] https://github.com/STEllAR-GROUP/hpx/issues/1291
[3609] https://github.com/STEllAR-GROUP/hpx/issues/1290
[3610] https://github.com/STEllAR-GROUP/hpx/issues/1289
[3611] https://github.com/STEllAR-GROUP/hpx/issues/1288
[3612] https://github.com/STEllAR-GROUP/hpx/issues/1287
[3613] https://github.com/STEllAR-GROUP/hpx/issues/1286
[3614] https://github.com/STEllAR-GROUP/hpx/issues/1285
[3615] https://github.com/STEllAR-GROUP/hpx/issues/1284
[3616] https://github.com/STEllAR-GROUP/hpx/issues/1283
[3617] https://github.com/STEllAR-GROUP/hpx/issues/1282
[3618] https://github.com/STEllAR-GROUP/hpx/issues/1281
[3619] https://github.com/STEllAR-GROUP/hpx/issues/1280
[3620] https://github.com/STEllAR-GROUP/hpx/issues/1279
[3621] https://github.com/STEllAR-GROUP/hpx/issues/1278
[3622] https://github.com/STEllAR-GROUP/hpx/issues/1277

- Issue #1276[3623] - Remove merging map
- Issue #1274[3624] - Weird (wrong) string code in papi.cpp
- Issue #1273[3625] - Sequential task execution policy
- Issue #1272[3626] - Avoid CMake name clash for Boost.Thread library
- Issue #1271[3627] - Updates on HPX Test Units
- Issue #1270[3628] - hpx/util/safe_lexical_cast.hpp is added
- Issue #1269[3629] - Added default value for "LIB" cmake variable
- Issue #1268[3630] - Memory Counters not working
- Issue #1266[3631] - FindHPX.cmake is not installed
- Issue #1263[3632] - apply_remote test takes too long
- Issue #1262[3633] - Chrono cleanup
- Issue #1261[3634] - Need make install for papi counters and this builds all the examples
- Issue #1260[3635] - Documentation of Stencil example claims
- Issue #1259[3636] - Avoid double-linking Boost on Windows
- Issue #1257[3637] - Adding additional parameter to create_thread
- Issue #1256[3638] - added buildbot changes to release notes
- Issue #1255[3639] - Cannot build MiniGhost
- Issue #1253[3640] - hpx::thread defects
- Issue #1252[3641] - HPX_PREFIX is too fragile
- Issue #1250[3642] - switch_to_fiber_emulation does not work properly
- Issue #1249[3643] - Documentation is generated under Release folder
- Issue #1248[3644] - Fix usage of hpx_generic_coroutine_context and get tests passing on powerpc
- Issue #1247[3645] - Dynamic linking error

3623 https://github.com/STEllAR-GROUP/hpx/issues/1276
3624 https://github.com/STEllAR-GROUP/hpx/issues/1274
3625 https://github.com/STEllAR-GROUP/hpx/issues/1273
3626 https://github.com/STEllAR-GROUP/hpx/issues/1272
3627 https://github.com/STEllAR-GROUP/hpx/issues/1271
3628 https://github.com/STEllAR-GROUP/hpx/issues/1270
3629 https://github.com/STEllAR-GROUP/hpx/issues/1269
3630 https://github.com/STEllAR-GROUP/hpx/issues/1268
3631 https://github.com/STEllAR-GROUP/hpx/issues/1266
3632 https://github.com/STEllAR-GROUP/hpx/issues/1263
3633 https://github.com/STEllAR-GROUP/hpx/issues/1262
3634 https://github.com/STEllAR-GROUP/hpx/issues/1261
3635 https://github.com/STEllAR-GROUP/hpx/issues/1260
3636 https://github.com/STEllAR-GROUP/hpx/issues/1259
3637 https://github.com/STEllAR-GROUP/hpx/issues/1257
3638 https://github.com/STEllAR-GROUP/hpx/issues/1256
3639 https://github.com/STEllAR-GROUP/hpx/issues/1255
3640 https://github.com/STEllAR-GROUP/hpx/issues/1253
3641 https://github.com/STEllAR-GROUP/hpx/issues/1252
3642 https://github.com/STEllAR-GROUP/hpx/issues/1250
3643 https://github.com/STEllAR-GROUP/hpx/issues/1249
3644 https://github.com/STEllAR-GROUP/hpx/issues/1248
3645 https://github.com/STEllAR-GROUP/hpx/issues/1247

- Issue #1246[3646] - Make cpuid.cpp C++11 compliant
- Issue #1245[3647] - HPX fails on startup (setting thread affinity mask)
- Issue #1244[3648] - HPX_WITH_RDTSC configure test fails, but should succeed
- Issue #1243[3649] - CTest dashboard info for CSCS CDash drop location
- Issue #1242[3650] - Mac fixes
- Issue #1241[3651] - Failure in Distributed with Boost 1.56
- Issue #1240[3652] - fix a race condition in examples.diskperf
- Issue #1239[3653] - fix wait_each in examples.diskperf
- Issue #1238[3654] - Fixed #1237: hpx::util::portable_binary_iarchive failed
- Issue #1237[3655] - hpx::util::portable_binary_iarchive faileds
- Issue #1235[3656] - Fixing clang warnings and errors
- Issue #1234[3657] - TCP runs fail: Transport endpoint is not connected
- Issue #1233[3658] - Making sure the correct number of threads is registered with AGAS
- Issue #1232[3659] - Fixing race in wait_xxx
- Issue #1231[3660] - Parallel minmax
- Issue #1230[3661] - Distributed run of 1d_stencil_8 uses less threads than spec. & sometimes gives errors
- Issue #1229[3662] - Unstable number of threads
- Issue #1228[3663] - HPX link error (cmake / MPI)
- Issue #1226[3664] - Warning about struct/class thread_counters
- Issue #1225[3665] - Adding parallel::replace etc
- Issue #1224[3666] - Extending dataflow to pass through non-future arguments
- Issue #1223[3667] - Remaining find algorithms implemented, N4071
- Issue #1222[3668] - Merging all the changes

3646 https://github.com/STEllAR-GROUP/hpx/issues/1246
3647 https://github.com/STEllAR-GROUP/hpx/issues/1245
3648 https://github.com/STEllAR-GROUP/hpx/issues/1244
3649 https://github.com/STEllAR-GROUP/hpx/issues/1243
3650 https://github.com/STEllAR-GROUP/hpx/issues/1242
3651 https://github.com/STEllAR-GROUP/hpx/issues/1241
3652 https://github.com/STEllAR-GROUP/hpx/issues/1240
3653 https://github.com/STEllAR-GROUP/hpx/issues/1239
3654 https://github.com/STEllAR-GROUP/hpx/issues/1238
3655 https://github.com/STEllAR-GROUP/hpx/issues/1237
3656 https://github.com/STEllAR-GROUP/hpx/issues/1235
3657 https://github.com/STEllAR-GROUP/hpx/issues/1234
3658 https://github.com/STEllAR-GROUP/hpx/issues/1233
3659 https://github.com/STEllAR-GROUP/hpx/issues/1232
3660 https://github.com/STEllAR-GROUP/hpx/issues/1231
3661 https://github.com/STEllAR-GROUP/hpx/issues/1230
3662 https://github.com/STEllAR-GROUP/hpx/issues/1229
3663 https://github.com/STEllAR-GROUP/hpx/issues/1228
3664 https://github.com/STEllAR-GROUP/hpx/issues/1226
3665 https://github.com/STEllAR-GROUP/hpx/issues/1225
3666 https://github.com/STEllAR-GROUP/hpx/issues/1224
3667 https://github.com/STEllAR-GROUP/hpx/issues/1223
3668 https://github.com/STEllAR-GROUP/hpx/issues/1222

- Issue #1221[3669] - No error output when using mpirun with hpx
- Issue #1219[3670] - Adding new AGAS cache performance counters
- Issue #1216[3671] - Fixing using futures (clients) as arguments to actions
- Issue #1215[3672] - Error compiling simple component
- Issue #1214[3673] - Stencil docs
- Issue #1213[3674] - Using more than a few dozen MPI processes on SuperMike results in a seg fault before getting to hpx_main
- Issue #1212[3675] - Parallel rotate
- Issue #1211[3676] - Direct actions cause the future's shared_state to be leaked
- Issue #1210[3677] - Refactored local::promise to be standard conformant
- Issue #1209[3678] - Improve command line handling
- Issue #1208[3679] - Adding parallel::reverse and parallel::reverse_copy
- Issue #1207[3680] - Add copy_backward and move_backward
- Issue #1206[3681] - N4071 additional algorithms implemented
- Issue #1204[3682] - Cmake simplification and various other minor changes
- Issue #1203[3683] - Implementing new launch policy for (local) async: `hpx::launch::fork`.
- Issue #1202[3684] - Failed assertion in connection_cache.hpp
- Issue #1201[3685] - pkg-config doesn't add mpi link directories
- Issue #1200[3686] - Error when querying time performance counters
- Issue #1199[3687] - library path is now configurable (again)
- Issue #1198[3688] - Error when querying performance counters
- Issue #1197[3689] - tests fail with intel compiler
- Issue #1196[3690] - Silence several warnings
- Issue #1195[3691] - Rephrase initializers to work with VC++ 2012

[3669] https://github.com/STEllAR-GROUP/hpx/issues/1221
[3670] https://github.com/STEllAR-GROUP/hpx/issues/1219
[3671] https://github.com/STEllAR-GROUP/hpx/issues/1216
[3672] https://github.com/STEllAR-GROUP/hpx/issues/1215
[3673] https://github.com/STEllAR-GROUP/hpx/issues/1214
[3674] https://github.com/STEllAR-GROUP/hpx/issues/1213
[3675] https://github.com/STEllAR-GROUP/hpx/issues/1212
[3676] https://github.com/STEllAR-GROUP/hpx/issues/1211
[3677] https://github.com/STEllAR-GROUP/hpx/issues/1210
[3678] https://github.com/STEllAR-GROUP/hpx/issues/1209
[3679] https://github.com/STEllAR-GROUP/hpx/issues/1208
[3680] https://github.com/STEllAR-GROUP/hpx/issues/1207
[3681] https://github.com/STEllAR-GROUP/hpx/issues/1206
[3682] https://github.com/STEllAR-GROUP/hpx/issues/1204
[3683] https://github.com/STEllAR-GROUP/hpx/issues/1203
[3684] https://github.com/STEllAR-GROUP/hpx/issues/1202
[3685] https://github.com/STEllAR-GROUP/hpx/issues/1201
[3686] https://github.com/STEllAR-GROUP/hpx/issues/1200
[3687] https://github.com/STEllAR-GROUP/hpx/issues/1199
[3688] https://github.com/STEllAR-GROUP/hpx/issues/1198
[3689] https://github.com/STEllAR-GROUP/hpx/issues/1197
[3690] https://github.com/STEllAR-GROUP/hpx/issues/1196
[3691] https://github.com/STEllAR-GROUP/hpx/issues/1195

- Issue #1194[3692] - Simplify parallel algorithms
- Issue #1193[3693] - Adding `parallel::equal`
- Issue #1192[3694] - HPX(out_of_memory) on including <hpx/hpx.hpp>
- Issue #1191[3695] - Fixing #1189
- Issue #1190[3696] - Chrono cleanup
- Issue #1189[3697] - Deadlock .. somewhere? (probably serialization)
- Issue #1188[3698] - Removed `future::get_status()`
- Issue #1186[3699] - Fixed FindOpenCL to find current AMD APP SDK
- Issue #1184[3700] - Tweaking future unwrapping
- Issue #1183[3701] - Extended `parallel::reduce`
- Issue #1182[3702] - `future::unwrap` hangs for `launch::deferred`
- Issue #1181[3703] - Adding `all_of`, `any_of`, and `none_of` and corresponding documentation
- Issue #1180[3704] - `hpx::cout` defect
- Issue #1179[3705] - `hpx::async` does not work for member function pointers when called on types with self-defined unary `operator*`
- Issue #1178[3706] - Implemented variadic `hpx::util::zip_iterator`
- Issue #1177[3707] - MPI parcelport defect
- Issue #1176[3708] - `HPX_DEFINE_COMPONENT_CONST_ACTION_TPL` does not have a 2-argument version
- Issue #1175[3709] - Create util::zip_iterator working with util::tuple<>
- Issue #1174[3710] - Error Building HPX on linux, root_certificate_authority.cpp
- Issue #1173[3711] - hpx::cout output lost
- Issue #1172[3712] - HPX build error with Clang 3.4.2
- Issue #1171[3713] - `CMAKE_INSTALL_PREFIX` ignored
- Issue #1170[3714] - Close hpx_benchmarks repository on Github

[3692] https://github.com/STEllAR-GROUP/hpx/issues/1194
[3693] https://github.com/STEllAR-GROUP/hpx/issues/1193
[3694] https://github.com/STEllAR-GROUP/hpx/issues/1192
[3695] https://github.com/STEllAR-GROUP/hpx/issues/1191
[3696] https://github.com/STEllAR-GROUP/hpx/issues/1190
[3697] https://github.com/STEllAR-GROUP/hpx/issues/1189
[3698] https://github.com/STEllAR-GROUP/hpx/issues/1188
[3699] https://github.com/STEllAR-GROUP/hpx/issues/1186
[3700] https://github.com/STEllAR-GROUP/hpx/issues/1184
[3701] https://github.com/STEllAR-GROUP/hpx/issues/1183
[3702] https://github.com/STEllAR-GROUP/hpx/issues/1182
[3703] https://github.com/STEllAR-GROUP/hpx/issues/1181
[3704] https://github.com/STEllAR-GROUP/hpx/issues/1180
[3705] https://github.com/STEllAR-GROUP/hpx/issues/1179
[3706] https://github.com/STEllAR-GROUP/hpx/issues/1178
[3707] https://github.com/STEllAR-GROUP/hpx/issues/1177
[3708] https://github.com/STEllAR-GROUP/hpx/issues/1176
[3709] https://github.com/STEllAR-GROUP/hpx/issues/1175
[3710] https://github.com/STEllAR-GROUP/hpx/issues/1174
[3711] https://github.com/STEllAR-GROUP/hpx/issues/1173
[3712] https://github.com/STEllAR-GROUP/hpx/issues/1172
[3713] https://github.com/STEllAR-GROUP/hpx/issues/1171
[3714] https://github.com/STEllAR-GROUP/hpx/issues/1170

- Issue #1169[3715] - Buildbot emails have syntax error in url
- Issue #1167[3716] - Merge partial implementation of standards proposal N3960
- Issue #1166[3717] - Fixed several compiler warnings
- Issue #1165[3718] - cmake warns: "tests.regressions.actions" does not exist
- Issue #1164[3719] - Want my own serialization of hpx::future
- Issue #1162[3720] - Segfault in hello_world example
- Issue #1161[3721] - Use `HPX_ASSERT` to aid the compiler
- Issue #1160[3722] - Do not put -DNDEBUG into hpx_application.pc
- Issue #1159[3723] - Support Clang 3.4.2
- Issue #1158[3724] - Fixed #1157: Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1157[3725] - Rename when_n/wait_n, add when_xxx_n/wait_xxx_n
- Issue #1156[3726] - Force inlining fails
- Issue #1155[3727] - changed header of printout to be compatible with python csv module
- Issue #1154[3728] - Fixing iostreams
- Issue #1153[3729] - Standard manipulators (like std::endl) do not work with hpx::ostream
- Issue #1152[3730] - Functions revamp
- Issue #1151[3731] - Suppressing cmake 3.0 policy warning for CMP0026
- Issue #1150[3732] - Client Serialization error
- Issue #1149[3733] - Segfault on Stampede
- Issue #1148[3734] - Refactoring mini-ghost
- Issue #1147[3735] - N3960 copy_if and copy_n implemented and tested
- Issue #1146[3736] - Stencil print
- Issue #1145[3737] - N3960 hpx::parallel::copy implemented and tested

3715 https://github.com/STEllAR-GROUP/hpx/issues/1169
3716 https://github.com/STEllAR-GROUP/hpx/issues/1167
3717 https://github.com/STEllAR-GROUP/hpx/issues/1166
3718 https://github.com/STEllAR-GROUP/hpx/issues/1165
3719 https://github.com/STEllAR-GROUP/hpx/issues/1164
3720 https://github.com/STEllAR-GROUP/hpx/issues/1162
3721 https://github.com/STEllAR-GROUP/hpx/issues/1161
3722 https://github.com/STEllAR-GROUP/hpx/issues/1160
3723 https://github.com/STEllAR-GROUP/hpx/issues/1159
3724 https://github.com/STEllAR-GROUP/hpx/issues/1158
3725 https://github.com/STEllAR-GROUP/hpx/issues/1157
3726 https://github.com/STEllAR-GROUP/hpx/issues/1156
3727 https://github.com/STEllAR-GROUP/hpx/issues/1155
3728 https://github.com/STEllAR-GROUP/hpx/issues/1154
3729 https://github.com/STEllAR-GROUP/hpx/issues/1153
3730 https://github.com/STEllAR-GROUP/hpx/issues/1152
3731 https://github.com/STEllAR-GROUP/hpx/issues/1151
3732 https://github.com/STEllAR-GROUP/hpx/issues/1150
3733 https://github.com/STEllAR-GROUP/hpx/issues/1149
3734 https://github.com/STEllAR-GROUP/hpx/issues/1148
3735 https://github.com/STEllAR-GROUP/hpx/issues/1147
3736 https://github.com/STEllAR-GROUP/hpx/issues/1146
3737 https://github.com/STEllAR-GROUP/hpx/issues/1145

- Issue #1144[3738] - OpenMP examples 1d_stencil do not build
- Issue #1143[3739] - 1d_stencil OpenMP examples do not build
- Issue #1142[3740] - Cannot build HPX with gcc 4.6 on OS X
- Issue #1140[3741] - Fix OpenMP lookup, enable usage of config tests in external CMake projects.
- Issue #1139[3742] - hpx/hpx/config/compiler_specific.hpp
- Issue #1138[3743] - clean up pkg-config files
- Issue #1137[3744] - Improvements to create binary packages
- Issue #1136[3745] - HPX_GCC_VERSION not defined on all compilers
- Issue #1135[3746] - Avoiding collision between winsock2.h and windows.h
- Issue #1134[3747] - Making sure, that hpx::finalize can be called from any locality
- Issue #1133[3748] - 1d stencil examples
- Issue #1131[3749] - Refactor unique_function implementation
- Issue #1130[3750] - Unique function
- Issue #1129[3751] - Some fixes to the Build system on OS X
- Issue #1128[3752] - Action future args
- Issue #1127[3753] - Executor causes segmentation fault
- Issue #1124[3754] - Adding new API functions: `register_id_with_basename`, `unregister_id_with_basename`, `find_ids_from_basename`; adding test
- Issue #1123[3755] - Reduce nesting of try-catch construct in `encode_parcels`?
- Issue #1122[3756] - Client base fixes
- Issue #1121[3757] - Update `hpxrun.py.in`
- Issue #1120[3758] - HTTS2 tests compile errors on v110 (VS2012)
- Issue #1119[3759] - Remove references to boost::atomic in accumulator example
- Issue #1118[3760] - Only build test thread_pool_executor_1114_test if `HPX_LOCAL_SCHEDULER` is set

3738 https://github.com/STEllAR-GROUP/hpx/issues/1144
3739 https://github.com/STEllAR-GROUP/hpx/issues/1143
3740 https://github.com/STEllAR-GROUP/hpx/issues/1142
3741 https://github.com/STEllAR-GROUP/hpx/issues/1140
3742 https://github.com/STEllAR-GROUP/hpx/issues/1139
3743 https://github.com/STEllAR-GROUP/hpx/issues/1138
3744 https://github.com/STEllAR-GROUP/hpx/issues/1137
3745 https://github.com/STEllAR-GROUP/hpx/issues/1136
3746 https://github.com/STEllAR-GROUP/hpx/issues/1135
3747 https://github.com/STEllAR-GROUP/hpx/issues/1134
3748 https://github.com/STEllAR-GROUP/hpx/issues/1133
3749 https://github.com/STEllAR-GROUP/hpx/issues/1131
3750 https://github.com/STEllAR-GROUP/hpx/issues/1130
3751 https://github.com/STEllAR-GROUP/hpx/issues/1129
3752 https://github.com/STEllAR-GROUP/hpx/issues/1128
3753 https://github.com/STEllAR-GROUP/hpx/issues/1127
3754 https://github.com/STEllAR-GROUP/hpx/issues/1124
3755 https://github.com/STEllAR-GROUP/hpx/issues/1123
3756 https://github.com/STEllAR-GROUP/hpx/issues/1122
3757 https://github.com/STEllAR-GROUP/hpx/issues/1121
3758 https://github.com/STEllAR-GROUP/hpx/issues/1120
3759 https://github.com/STEllAR-GROUP/hpx/issues/1119
3760 https://github.com/STEllAR-GROUP/hpx/issues/1118

- Issue #1117[3761] - local_queue_executor linker error on vc110
- Issue #1116[3762] - Disabled performance counter should give runtime errors, not invalid data
- Issue #1115[3763] - Compile error with Intel C++ 13.1
- Issue #1114[3764] - Default constructed executor is not usable
- Issue #1113[3765] - Fast compilation of logging causes ABI incompatibilities between different `NDEBUG` values
- Issue #1112[3766] - Using thread_pool_executors causes segfault
- Issue #1111[3767] - `hpx::threads::get_thread_data` always returns zero
- Issue #1110[3768] - Remove unnecessary null pointer checks
- Issue #1109[3769] - More tests adjustments
- Issue #1108[3770] - Clarify build rules for "libboost_atomic-mt.so"?
- Issue #1107[3771] - Remove unnecessary null pointer checks
- Issue #1106[3772] - network_storage benchmark imporvements, adding legends to plots and tidying layout
- Issue #1105[3773] - Add more plot outputs and improve instructions doc
- Issue #1104[3774] - Complete quoting for parameters of some CMake commands
- Issue #1103[3775] - Work on test/scripts
- Issue #1102[3776] - Changed minimum requirement of window install to 2012
- Issue #1101[3777] - Changed minimum requirement of window install to 2012
- Issue #1100[3778] - Changed readme to no longer specify using MSVC 2010 compiler
- Issue #1099[3779] - Error returning futures from component actions
- Issue #1098[3780] - Improve storage test
- Issue #1097[3781] - data_actions quickstart example calls missing function decorate_action of data_get_action
- Issue #1096[3782] - MPI parcelport broken with new zero copy optimization
- Issue #1095[3783] - Warning C4005: _WIN32_WINNT: Macro redefinition

[3761] https://github.com/STEllAR-GROUP/hpx/issues/1117
[3762] https://github.com/STEllAR-GROUP/hpx/issues/1116
[3763] https://github.com/STEllAR-GROUP/hpx/issues/1115
[3764] https://github.com/STEllAR-GROUP/hpx/issues/1114
[3765] https://github.com/STEllAR-GROUP/hpx/issues/1113
[3766] https://github.com/STEllAR-GROUP/hpx/issues/1112
[3767] https://github.com/STEllAR-GROUP/hpx/issues/1111
[3768] https://github.com/STEllAR-GROUP/hpx/issues/1110
[3769] https://github.com/STEllAR-GROUP/hpx/issues/1109
[3770] https://github.com/STEllAR-GROUP/hpx/issues/1108
[3771] https://github.com/STEllAR-GROUP/hpx/issues/1107
[3772] https://github.com/STEllAR-GROUP/hpx/issues/1106
[3773] https://github.com/STEllAR-GROUP/hpx/issues/1105
[3774] https://github.com/STEllAR-GROUP/hpx/issues/1104
[3775] https://github.com/STEllAR-GROUP/hpx/issues/1103
[3776] https://github.com/STEllAR-GROUP/hpx/issues/1102
[3777] https://github.com/STEllAR-GROUP/hpx/issues/1101
[3778] https://github.com/STEllAR-GROUP/hpx/issues/1100
[3779] https://github.com/STEllAR-GROUP/hpx/issues/1099
[3780] https://github.com/STEllAR-GROUP/hpx/issues/1098
[3781] https://github.com/STEllAR-GROUP/hpx/issues/1097
[3782] https://github.com/STEllAR-GROUP/hpx/issues/1096
[3783] https://github.com/STEllAR-GROUP/hpx/issues/1095

- Issue #1094[3784] - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS in master
- Issue #1093[3785] - Syntax error for -DHPX_UNIQUE_FUTURE_ALIAS
- Issue #1092[3786] - Rename unique_future<> back to future<>
- Issue #1091[3787] - Inconsistent error message
- Issue #1090[3788] - On windows 8.1 the examples crashed if using more than one os thread
- Issue #1089[3789] - Components should be allowed to have their own executor
- Issue #1088[3790] - Add possibility to select a network interface for the ibverbs parcelport
- Issue #1087[3791] - ibverbs and ipc parcelport uses zero copy optimization
- Issue #1083[3792] - Make shell examples copyable in docs
- Issue #1082[3793] - Implement proper termination detection during shutdown
- Issue #1081[3794] - Implement thread_specific_ptr for hpx::threads
- Issue #1072[3795] - make install not working properly
- Issue #1070[3796] - Complete quoting for parameters of some CMake commands
- Issue #1059[3797] - Fix more unused variable warnings
- Issue #1051[3798] - Implement when_each
- Issue #973[3799] - Would like option to report hwloc bindings
- Issue #970[3800] - Bad flags for Fortran compiler
- Issue #941[3801] - Create a proper user level context switching class for BG/Q
- Issue #935[3802] - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #934[3803] - Want to build HPX without dynamic libraries
- Issue #927[3804] - Make hpx/lcos/reduce.hpp accept futures of id_type
- Issue #926[3805] - All unit tests that are run with more than one thread with CTest/hpx_run_test should configure hpx.os_threads
- Issue #925[3806] - regression_dataflow_791 needs to be brought in line with HPX standards

[3784] https://github.com/STEllAR-GROUP/hpx/issues/1094
[3785] https://github.com/STEllAR-GROUP/hpx/issues/1093
[3786] https://github.com/STEllAR-GROUP/hpx/issues/1092
[3787] https://github.com/STEllAR-GROUP/hpx/issues/1091
[3788] https://github.com/STEllAR-GROUP/hpx/issues/1090
[3789] https://github.com/STEllAR-GROUP/hpx/issues/1089
[3790] https://github.com/STEllAR-GROUP/hpx/issues/1088
[3791] https://github.com/STEllAR-GROUP/hpx/issues/1087
[3792] https://github.com/STEllAR-GROUP/hpx/issues/1083
[3793] https://github.com/STEllAR-GROUP/hpx/issues/1082
[3794] https://github.com/STEllAR-GROUP/hpx/issues/1081
[3795] https://github.com/STEllAR-GROUP/hpx/issues/1072
[3796] https://github.com/STEllAR-GROUP/hpx/issues/1070
[3797] https://github.com/STEllAR-GROUP/hpx/issues/1059
[3798] https://github.com/STEllAR-GROUP/hpx/issues/1051
[3799] https://github.com/STEllAR-GROUP/hpx/issues/973
[3800] https://github.com/STEllAR-GROUP/hpx/issues/970
[3801] https://github.com/STEllAR-GROUP/hpx/issues/941
[3802] https://github.com/STEllAR-GROUP/hpx/issues/935
[3803] https://github.com/STEllAR-GROUP/hpx/issues/934
[3804] https://github.com/STEllAR-GROUP/hpx/issues/927
[3805] https://github.com/STEllAR-GROUP/hpx/issues/926
[3806] https://github.com/STEllAR-GROUP/hpx/issues/925

- Issue #899[3807] - Fix race conditions in regression tests
- Issue #879[3808] - Hung test leads to cascading test failure; make tests should support the MPI parcelport
- Issue #865[3809] - future<T> and friends shall work for movable only Ts
- Issue #847[3810] - Dynamic libraries are not installed on OS X
- Issue #816[3811] - First Program tutorial pull request
- Issue #799[3812] - Wrap lexical_cast to avoid exceptions
- Issue #720[3813] - broken configuration when using ccmake on Ubuntu
- Issue #622[3814] - `--hpx:hpx` and `--hpx:debug-hpx-log` is nonsensical
- Issue #525[3815] - Extend barrier LCO test to run in distributed
- Issue #515[3816] - Multi-destination version of hpx::apply is broken
- Issue #509[3817] - Push Boost.Atomic changes upstream
- Issue #503[3818] - Running HPX applications on Windows should not require setting %PATH%
- Issue #461[3819] - Add a compilation sanity test
- Issue #456[3820] - hpx_run_tests.py should log output from tests that timeout
- Issue #454[3821] - Investigate threadmanager performance
- Issue #345[3822] - Add more versatile environmental/cmake variable support to hpx_find_* CMake macros
- Issue #209[3823] - Support multiple configurations in generated build files
- Issue #190[3824] - hpx::cout should be a std::ostream
- Issue #189[3825] - iostreams component should use startup/shutdown functions
- Issue #183[3826] - Use Boost.ICL for correctness in AGAS
- Issue #44[3827] - Implement real futures

[3807] https://github.com/STEllAR-GROUP/hpx/issues/899
[3808] https://github.com/STEllAR-GROUP/hpx/issues/879
[3809] https://github.com/STEllAR-GROUP/hpx/issues/865
[3810] https://github.com/STEllAR-GROUP/hpx/issues/847
[3811] https://github.com/STEllAR-GROUP/hpx/issues/816
[3812] https://github.com/STEllAR-GROUP/hpx/issues/799
[3813] https://github.com/STEllAR-GROUP/hpx/issues/720
[3814] https://github.com/STEllAR-GROUP/hpx/issues/622
[3815] https://github.com/STEllAR-GROUP/hpx/issues/525
[3816] https://github.com/STEllAR-GROUP/hpx/issues/515
[3817] https://github.com/STEllAR-GROUP/hpx/issues/509
[3818] https://github.com/STEllAR-GROUP/hpx/issues/503
[3819] https://github.com/STEllAR-GROUP/hpx/issues/461
[3820] https://github.com/STEllAR-GROUP/hpx/issues/456
[3821] https://github.com/STEllAR-GROUP/hpx/issues/454
[3822] https://github.com/STEllAR-GROUP/hpx/issues/345
[3823] https://github.com/STEllAR-GROUP/hpx/issues/209
[3824] https://github.com/STEllAR-GROUP/hpx/issues/190
[3825] https://github.com/STEllAR-GROUP/hpx/issues/189
[3826] https://github.com/STEllAR-GROUP/hpx/issues/183
[3827] https://github.com/STEllAR-GROUP/hpx/issues/44

2.11.14 *HPX* V0.9.8 (Mar 24, 2014)

We have had over 800 commits since the last release and we have closed over 65 tickets (bugs, feature requests, etc.).

With the changes below, *HPX* is once again leading the charge of a whole new era of computation. By intrinsically breaking down and synchronizing the work to be done, *HPX* insures that application developers will no longer have to fret about where a segment of code executes. That allows coders to focus their time and energy to understanding the data dependencies of their algorithms and thereby the core obstacles to an efficient code. Here are some of the advantages of using *HPX*:

- *HPX* is solidly rooted in a sophisticated theoretical execution model – ParalleX
- *HPX* exposes an API fully conforming to the C++11 and the draft C++14 standards, extended and applied to distributed computing. Everything programmers know about the concurrency primitives of the standard C++ library is still valid in the context of *HPX*.
- It provides a competitive, high performance implementation of modern, future-proof ideas which gives an smooth migration path from today's mainstream techniques
- There is no need for the programmer to worry about lower level parallelization paradigms like threads or message passing; no need to understand pthreads, MPI, OpenMP, or Windows threads, etc.
- There is no need to think about different types of parallelism such as tasks, pipelines, or fork-join, task or data parallelism.
- The same source of your program compiles and runs on Linux, BlueGene/Q, Mac OS X, Windows, and Android.
- The same code runs on shared memory multi-core systems and supercomputers, on handheld devices and Intel® Xeon Phi™ accelerators, or a heterogeneous mix of those.

General changes

- A major API breaking change for this release was introduced by implementing `hpx::future` and `hpx::shared_future` fully in conformance with the C++11 Standard[3828]. 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[3829]. 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[3830].
- 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.

[3828] http://www.open-std.org/jtc1/sc22/wg21
[3829] http://mail.cct.lsu.edu/pipermail/hpx-users/2014-January/000141.html
[3830] http://www.open-std.org/jtc1/sc22/wg21

- We added new API functionality like `hpx::migrate` and `hpx::copy_component` which are the basic building blocks necessary for implementing higher level abstractions for system-wide load balancing, runtime-adaptive resource management, and object-oriented checkpointing and state-management.
- We removed the use of C++11 move emulation (using Boost.Move), replacing it with C++11 rvalue references. This is the first step towards using more and more native C++11 facilities which we plan to introduce in the future.
- We improved the reference counting scheme used by *HPX* which helps managing distributed objects and memory. This improves the overall stability of *HPX* and further simplifies writing real world applications.
- The minimal Boost version required to use *HPX* is now V1.49.0.
- This release coincides with the first release of HPXPI (V0.1.0), the first implementation of the XPI specification[3831].

Bug fixes (closed tickets)

Here is a list of the important tickets we closed for this release.

- Issue #1086[3832] - Expose internal boost::shared_array to allow user management of array lifetime
- Issue #1083[3833] - Make shell examples copyable in docs
- Issue #1080[3834] - /threads{locality#*/total}/count/cumulative broken
- Issue #1079[3835] - Build problems on OS X
- Issue #1078[3836] - Improve robustness of component loading
- Issue #1077[3837] - Fix a missing enum definition for 'take' mode
- Issue #1076[3838] - Merge Jb master
- Issue #1075[3839] - Unknown CMake command "add_hpx_pseudo_target"
- Issue #1074[3840] - Implement `apply_continue_callback` and `apply_colocated_callback`
- Issue #1073[3841] - The new `apply_colocated` and `async_colocated` functions lead to automatic registered functions
- Issue #1071[3842] - Remove deferred_packaged_task
- Issue #1069[3843] - serialize_buffer with allocator fails at destruction
- Issue #1068[3844] - Coroutine include and forward declarations missing
- Issue #1067[3845] - Add allocator support to util::serialize_buffer

[3831] https://github.com/STEllAR-GROUP/hpxpi/blob/master/spec.pdf?raw=true
[3832] https://github.com/STEllAR-GROUP/hpx/issues/1086
[3833] https://github.com/STEllAR-GROUP/hpx/issues/1083
[3834] https://github.com/STEllAR-GROUP/hpx/issues/1080
[3835] https://github.com/STEllAR-GROUP/hpx/issues/1079
[3836] https://github.com/STEllAR-GROUP/hpx/issues/1078
[3837] https://github.com/STEllAR-GROUP/hpx/issues/1077
[3838] https://github.com/STEllAR-GROUP/hpx/issues/1076
[3839] https://github.com/STEllAR-GROUP/hpx/issues/1075
[3840] https://github.com/STEllAR-GROUP/hpx/issues/1074
[3841] https://github.com/STEllAR-GROUP/hpx/issues/1073
[3842] https://github.com/STEllAR-GROUP/hpx/issues/1071
[3843] https://github.com/STEllAR-GROUP/hpx/issues/1069
[3844] https://github.com/STEllAR-GROUP/hpx/issues/1068
[3845] https://github.com/STEllAR-GROUP/hpx/issues/1067

- Issue #1066[3846] - Allow for MPI_Init being called before HPX launches
- Issue #1065[3847] - AGAS cache isn't used/populated on worker localities
- Issue #1064[3848] - Reorder includes to ensure ws2 includes early
- Issue #1063[3849] - Add `hpx::runtime::suspend` and `hpx::runtime::resume`
- Issue #1062[3850] - Fix `async_continue` to properly handle return types
- Issue #1061[3851] - Implement `async_colocated` and `apply_colocated`
- Issue #1060[3852] - Implement minimal component migration
- Issue #1058[3853] - Remove `HPX_UTIL_TUPLE` from code base
- Issue #1057[3854] - Add performance counters for threading subsystem
- Issue #1055[3855] - Thread allocation uses two memory pools
- Issue #1053[3856] - Work stealing flawed
- Issue #1052[3857] - Fix a number of warnings
- Issue #1049[3858] - Fixes for TLS on OSX and more reliable test running
- Issue #1048[3859] - Fixing after 588 hang
- Issue #1047[3860] - Use port '0' for networking when using one locality
- Issue #1046[3861] - `composable_guard` test is broken when having more than one thread
- Issue #1045[3862] - Security missing headers
- Issue #1044[3863] - Native TLS on FreeBSD via __thread
- Issue #1043[3864] - `async` et.al. compute the wrong result type
- Issue #1042[3865] - `async` et.al. implicitly unwrap reference_wrappers
- Issue #1041[3866] - Remove redundant costly Kleene stars from regex searches
- Issue #1040[3867] - CMake script regex match patterns has unnecessary kleenes
- Issue #1039[3868] - Remove use of Boost.Move and replace with std::move and real rvalue refs

[3846] https://github.com/STEllAR-GROUP/hpx/issues/1066
[3847] https://github.com/STEllAR-GROUP/hpx/issues/1065
[3848] https://github.com/STEllAR-GROUP/hpx/issues/1064
[3849] https://github.com/STEllAR-GROUP/hpx/issues/1063
[3850] https://github.com/STEllAR-GROUP/hpx/issues/1062
[3851] https://github.com/STEllAR-GROUP/hpx/issues/1061
[3852] https://github.com/STEllAR-GROUP/hpx/issues/1060
[3853] https://github.com/STEllAR-GROUP/hpx/issues/1058
[3854] https://github.com/STEllAR-GROUP/hpx/issues/1057
[3855] https://github.com/STEllAR-GROUP/hpx/issues/1055
[3856] https://github.com/STEllAR-GROUP/hpx/issues/1053
[3857] https://github.com/STEllAR-GROUP/hpx/issues/1052
[3858] https://github.com/STEllAR-GROUP/hpx/issues/1049
[3859] https://github.com/STEllAR-GROUP/hpx/issues/1048
[3860] https://github.com/STEllAR-GROUP/hpx/issues/1047
[3861] https://github.com/STEllAR-GROUP/hpx/issues/1046
[3862] https://github.com/STEllAR-GROUP/hpx/issues/1045
[3863] https://github.com/STEllAR-GROUP/hpx/issues/1044
[3864] https://github.com/STEllAR-GROUP/hpx/issues/1043
[3865] https://github.com/STEllAR-GROUP/hpx/issues/1042
[3866] https://github.com/STEllAR-GROUP/hpx/issues/1041
[3867] https://github.com/STEllAR-GROUP/hpx/issues/1040
[3868] https://github.com/STEllAR-GROUP/hpx/issues/1039

- Issue #1038[3869] - Bump minimal required Boost to 1.49.0
- Issue #1037[3870] - Implicit unwrapping of futures in async broken
- Issue #1036[3871] - Scheduler hangs when user code attempts to "block" OS-threads
- Issue #1035[3872] - Idle-rate counter always reports 100% idle rate
- Issue #1034[3873] - Symbolic name registration causes application hangs
- Issue #1033[3874] - Application options read in from an options file generate an error message
- Issue #1032[3875] - `hpx::id_type` local reference counting is wrong
- Issue #1031[3876] - Negative entry in reference count table
- Issue #1030[3877] - Implement condition_variable
- Issue #1029[3878] - Deadlock in thread scheduling subsystem
- Issue #1028[3879] - HPX-thread cumulative count performance counters report incorrect value
- Issue #1027[3880] - Expose `hpx::thread_interrupted` error code as a separate exception type
- Issue #1026[3881] - Exceptions thrown in asynchronous calls can be lost if the value of the future is never queried
- Issue #1025[3882] - `future::wait_for/wait_until` do not remove callback
- Issue #1024[3883] - Remove dependence to boost assert and create hpx assert
- Issue #1023[3884] - Segfaults with tcmalloc
- Issue #1022[3885] - prerequisites link in readme is broken
- Issue #1020[3886] - HPX Deadlock on external synchronization
- Issue #1019[3887] - Convert using `BOOST_ASSERT` to `HPX_ASSERT`
- Issue #1018[3888] - compiling bug with gcc 4.8.1
- Issue #1017[3889] - Possible crash in io_pool executor
- Issue #1016[3890] - Crash at startup
- Issue #1014[3891] - Implement Increment/Decrement Merging

[3869] https://github.com/STEllAR-GROUP/hpx/issues/1038
[3870] https://github.com/STEllAR-GROUP/hpx/issues/1037
[3871] https://github.com/STEllAR-GROUP/hpx/issues/1036
[3872] https://github.com/STEllAR-GROUP/hpx/issues/1035
[3873] https://github.com/STEllAR-GROUP/hpx/issues/1034
[3874] https://github.com/STEllAR-GROUP/hpx/issues/1033
[3875] https://github.com/STEllAR-GROUP/hpx/issues/1032
[3876] https://github.com/STEllAR-GROUP/hpx/issues/1031
[3877] https://github.com/STEllAR-GROUP/hpx/issues/1030
[3878] https://github.com/STEllAR-GROUP/hpx/issues/1029
[3879] https://github.com/STEllAR-GROUP/hpx/issues/1028
[3880] https://github.com/STEllAR-GROUP/hpx/issues/1027
[3881] https://github.com/STEllAR-GROUP/hpx/issues/1026
[3882] https://github.com/STEllAR-GROUP/hpx/issues/1025
[3883] https://github.com/STEllAR-GROUP/hpx/issues/1024
[3884] https://github.com/STEllAR-GROUP/hpx/issues/1023
[3885] https://github.com/STEllAR-GROUP/hpx/issues/1022
[3886] https://github.com/STEllAR-GROUP/hpx/issues/1020
[3887] https://github.com/STEllAR-GROUP/hpx/issues/1019
[3888] https://github.com/STEllAR-GROUP/hpx/issues/1018
[3889] https://github.com/STEllAR-GROUP/hpx/issues/1017
[3890] https://github.com/STEllAR-GROUP/hpx/issues/1016
[3891] https://github.com/STEllAR-GROUP/hpx/issues/1014

- Issue #1013[3892] - Add more logging channels to enable greater control over logging granularity
- Issue #1012[3893] - `--hpx:debug-hpx-log` and `--hpx:debug-agas-log` lead to non-thread safe writes
- Issue #1011[3894] - After installation, running applications from the build/staging directory no longer works
- Issue #1010[3895] - Mergable decrement requests are not being merged
- Issue #1009[3896] - `--hpx:list-symbolic-names` crashes
- Issue #1007[3897] - Components are not properly destroyed
- Issue #1006[3898] - Segfault/hang in set_data
- Issue #1003[3899] - Performance counter naming issue
- Issue #982[3900] - Race condition during startup
- Issue #912[3901] - OS X: component type not found in map
- Issue #663[3902] - Create a buildbot slave based on Clang 3.2/OSX
- Issue #636[3903] - Expose `this_locality::apply<act>(p1, p2);` for local execution
- Issue #197[3904] - Add `--console=address` option for PBS runs
- Issue #175[3905] - Asynchronous AGAS API

2.11.15 *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

[3892] https://github.com/STEllAR-GROUP/hpx/issues/1013
[3893] https://github.com/STEllAR-GROUP/hpx/issues/1012
[3894] https://github.com/STEllAR-GROUP/hpx/issues/1011
[3895] https://github.com/STEllAR-GROUP/hpx/issues/1010
[3896] https://github.com/STEllAR-GROUP/hpx/issues/1009
[3897] https://github.com/STEllAR-GROUP/hpx/issues/1007
[3898] https://github.com/STEllAR-GROUP/hpx/issues/1006
[3899] https://github.com/STEllAR-GROUP/hpx/issues/1003
[3900] https://github.com/STEllAR-GROUP/hpx/issues/982
[3901] https://github.com/STEllAR-GROUP/hpx/issues/912
[3902] https://github.com/STEllAR-GROUP/hpx/issues/663
[3903] https://github.com/STEllAR-GROUP/hpx/issues/636
[3904] https://github.com/STEllAR-GROUP/hpx/issues/197
[3905] https://github.com/STEllAR-GROUP/hpx/issues/175

- 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[3906] - Allow one to disable array optimizations and zero copy optimizations for each parcelport
- Issue #1004[3907] - Generate new HPX logo image for the docs
- Issue #1002[3908] - If MPI parcelport is not available, running HPX under mpirun should fail
- Issue #1001[3909] - Zero copy serialization raises assert
- Issue #1000[3910] - Can't connect to a HPX application running with the MPI parcelport from a non MPI parcelport locality
- Issue #999[3911] - Optimize `hpx::when_n`
- Issue #998[3912] - Fixed const-correctness
- Issue #997[3913] - Making serialize_buffer::data() type save
- Issue #996[3914] - Memory leak in hpx::lcos::promise
- Issue #995[3915] - Race while registering pre-shutdown functions
- Issue #994[3916] - thread_rescheduling regression test does not compile
- Issue #992[3917] - Correct comments and messages
- Issue #991[3918] - setcap cap_sys_rawio=ep for power profiling causes an HPX application to abort
- Issue #989[3919] - Jacobi hangs during execution

[3906] https://github.com/STEllAR-GROUP/hpx/issues/1005
[3907] https://github.com/STEllAR-GROUP/hpx/issues/1004
[3908] https://github.com/STEllAR-GROUP/hpx/issues/1002
[3909] https://github.com/STEllAR-GROUP/hpx/issues/1001
[3910] https://github.com/STEllAR-GROUP/hpx/issues/1000
[3911] https://github.com/STEllAR-GROUP/hpx/issues/999
[3912] https://github.com/STEllAR-GROUP/hpx/issues/998
[3913] https://github.com/STEllAR-GROUP/hpx/issues/997
[3914] https://github.com/STEllAR-GROUP/hpx/issues/996
[3915] https://github.com/STEllAR-GROUP/hpx/issues/995
[3916] https://github.com/STEllAR-GROUP/hpx/issues/994
[3917] https://github.com/STEllAR-GROUP/hpx/issues/992
[3918] https://github.com/STEllAR-GROUP/hpx/issues/991
[3919] https://github.com/STEllAR-GROUP/hpx/issues/989

- Issue #988[3920] - multiple_init test is failing
- Issue #986[3921] - Can't call a function called "init" from "main" when using <hpx/hpx_main.hpp>
- Issue #984[3922] - Reference counting tests are failing
- Issue #983[3923] - thread_suspension_executor test fails
- Issue #980[3924] - Terminating HPX threads don't leave stack in virgin state
- Issue #979[3925] - Static scheduler not in documents
- Issue #978[3926] - Preprocessing limits are broken
- Issue #977[3927] - Make tests.regressions.lcos.future_hang_on_get shorter
- Issue #976[3928] - Wrong library order in pkgconfig
- Issue #975[3929] - Please reopen #963
- Issue #974[3930] - Option pu-offset ignored in fixing_588 branch
- Issue #972[3931] - Cannot use MKL with HPX
- Issue #969[3932] - Non-existent INI files requested on the command line via `--hpx:config` do not cause warnings or errors.
- Issue #968[3933] - Cannot build examples in fixing_588 branch
- Issue #967[3934] - Command line description of `--hpx:queuing` seems wrong
- Issue #966[3935] - `--hpx:print-bind` physical core numbers are wrong
- Issue #965[3936] - Deadlock when building in Release mode
- Issue #963[3937] - Not all worker threads are working
- Issue #962[3938] - Problem with SLURM integration
- Issue #961[3939] - `--hpx:print-bind` outputs incorrect information
- Issue #960[3940] - Fix cut and paste error in documentation of get_thread_priority
- Issue #959[3941] - Change link to boost.atomic in documentation to point to boost.org
- Issue #958[3942] - Undefined reference to intrusive_ptr_release

[3920] https://github.com/STEllAR-GROUP/hpx/issues/988
[3921] https://github.com/STEllAR-GROUP/hpx/issues/986
[3922] https://github.com/STEllAR-GROUP/hpx/issues/984
[3923] https://github.com/STEllAR-GROUP/hpx/issues/983
[3924] https://github.com/STEllAR-GROUP/hpx/issues/980
[3925] https://github.com/STEllAR-GROUP/hpx/issues/979
[3926] https://github.com/STEllAR-GROUP/hpx/issues/978
[3927] https://github.com/STEllAR-GROUP/hpx/issues/977
[3928] https://github.com/STEllAR-GROUP/hpx/issues/976
[3929] https://github.com/STEllAR-GROUP/hpx/issues/975
[3930] https://github.com/STEllAR-GROUP/hpx/issues/974
[3931] https://github.com/STEllAR-GROUP/hpx/issues/972
[3932] https://github.com/STEllAR-GROUP/hpx/issues/969
[3933] https://github.com/STEllAR-GROUP/hpx/issues/968
[3934] https://github.com/STEllAR-GROUP/hpx/issues/967
[3935] https://github.com/STEllAR-GROUP/hpx/issues/966
[3936] https://github.com/STEllAR-GROUP/hpx/issues/965
[3937] https://github.com/STEllAR-GROUP/hpx/issues/963
[3938] https://github.com/STEllAR-GROUP/hpx/issues/962
[3939] https://github.com/STEllAR-GROUP/hpx/issues/961
[3940] https://github.com/STEllAR-GROUP/hpx/issues/960
[3941] https://github.com/STEllAR-GROUP/hpx/issues/959
[3942] https://github.com/STEllAR-GROUP/hpx/issues/958

- Issue #957[3943] - Make tuple standard compliant
- Issue #956[3944] - Segfault with a3382fb
- Issue #955[3945] - `--hpx:nodes` and `--hpx:nodefiles` do not work with foreign nodes
- Issue #954[3946] - Make order of arguments for hpx::async and hpx::broadcast consistent
- Issue #953[3947] - Cannot use MKL with HPX
- Issue #952[3948] - `register_[pre_]shutdown_function` never throw
- Issue #951[3949] - Assert when number of threads is greater than hardware concurrency
- Issue #948[3950] - `HPX_HAVE_GENERIC_CONTEXT_COROUTINES` conflicts with `HPX_HAVE_FIBER_BASED_COROUTINES`
- Issue #947[3951] - Need MPI_THREAD_MULTIPLE for backward compatibility
- Issue #946[3952] - HPX does not call `MPI_Finalize`
- Issue #945[3953] - Segfault with `hpx::lcos::broadcast`
- Issue #944[3954] - OS X: assertion `pu_offset_ < hardware_concurrency` failed
- Issue #943[3955] - #include <hpx/hpx_main.hpp> does not work
- Issue #942[3956] - Make the BG/Q work with -O3
- Issue #940[3957] - Use separator when concatenating locality name
- Issue #939[3958] - Refactor MPI parcelport to use `MPI_Wait` instead of multiple `MPI_Test` calls
- Issue #938[3959] - Want to officially access `client_base::gid_`
- Issue #937[3960] - `client_base::gid_` should be private``
- Issue #936[3961] - Want doxygen-like source code index
- Issue #935[3962] - Build error with gcc 4.6 and Boost 1.54.0 on hpx trunk and 0.9.6
- Issue #933[3963] - Cannot build HPX with Boost 1.54.0
- Issue #932[3964] - Components are destructed too early
- Issue #931[3965] - Make HPX work on BG/Q

3943 https://github.com/STEllAR-GROUP/hpx/issues/957
3944 https://github.com/STEllAR-GROUP/hpx/issues/956
3945 https://github.com/STEllAR-GROUP/hpx/issues/955
3946 https://github.com/STEllAR-GROUP/hpx/issues/954
3947 https://github.com/STEllAR-GROUP/hpx/issues/953
3948 https://github.com/STEllAR-GROUP/hpx/issues/952
3949 https://github.com/STEllAR-GROUP/hpx/issues/951
3950 https://github.com/STEllAR-GROUP/hpx/issues/948
3951 https://github.com/STEllAR-GROUP/hpx/issues/947
3952 https://github.com/STEllAR-GROUP/hpx/issues/946
3953 https://github.com/STEllAR-GROUP/hpx/issues/945
3954 https://github.com/STEllAR-GROUP/hpx/issues/944
3955 https://github.com/STEllAR-GROUP/hpx/issues/943
3956 https://github.com/STEllAR-GROUP/hpx/issues/942
3957 https://github.com/STEllAR-GROUP/hpx/issues/940
3958 https://github.com/STEllAR-GROUP/hpx/issues/939
3959 https://github.com/STEllAR-GROUP/hpx/issues/938
3960 https://github.com/STEllAR-GROUP/hpx/issues/937
3961 https://github.com/STEllAR-GROUP/hpx/issues/936
3962 https://github.com/STEllAR-GROUP/hpx/issues/935
3963 https://github.com/STEllAR-GROUP/hpx/issues/933
3964 https://github.com/STEllAR-GROUP/hpx/issues/932
3965 https://github.com/STEllAR-GROUP/hpx/issues/931

- Issue #930[3966] - make git-docs is broken
- Issue #929[3967] - Generating index in docs broken
- Issue #928[3968] - Optimize `hpx::util::static_` for C++11 compilers supporting magic statics
- Issue #924[3969] - Make kill_process_tree (in process.py) more robust on Mac OSX
- Issue #923[3970] - Correct BLAS and RNPL cmake tests
- Issue #922[3971] - Cannot link against BLAS
- Issue #921[3972] - Implement `hpx::mem_fn`
- Issue #920[3973] - Output locality with `--hpx:print-bind`
- Issue #919[3974] - Correct grammar; simplify boolean expressions
- Issue #918[3975] - Link to hello_world.cpp is broken
- Issue #917[3976] - adapt cmake file to new boostbook version
- Issue #916[3977] - fix problem building documentation with xsltproc >= 1.1.27
- Issue #915[3978] - Add another TBBMalloc library search path
- Issue #914[3979] - Build problem with Intel compiler on Stampede (TACC)
- Issue #913[3980] - fix error messages in fibonacci examples
- Issue #911[3981] - Update OS X build instructions
- Issue #910[3982] - Want like to specify MPI_ROOT instead of compiler wrapper script
- Issue #909[3983] - Warning about void* arithmetic
- Issue #908[3984] - Buildbot for MIC is broken
- Issue #906[3985] - Can't use `--hpx:bind=balanced` with multiple MPI processes
- Issue #905[3986] - `--hpx:bind` documentation should describe full grammar
- Issue #904[3987] - Add hpx::lcos::fold and hpx::lcos::inverse_fold collective operation
- Issue #903[3988] - Add `hpx::when_any_swapped()`

[3966] https://github.com/STEllAR-GROUP/hpx/issues/930
[3967] https://github.com/STEllAR-GROUP/hpx/issues/929
[3968] https://github.com/STEllAR-GROUP/hpx/issues/928
[3969] https://github.com/STEllAR-GROUP/hpx/issues/924
[3970] https://github.com/STEllAR-GROUP/hpx/issues/923
[3971] https://github.com/STEllAR-GROUP/hpx/issues/922
[3972] https://github.com/STEllAR-GROUP/hpx/issues/921
[3973] https://github.com/STEllAR-GROUP/hpx/issues/920
[3974] https://github.com/STEllAR-GROUP/hpx/issues/919
[3975] https://github.com/STEllAR-GROUP/hpx/issues/918
[3976] https://github.com/STEllAR-GROUP/hpx/issues/917
[3977] https://github.com/STEllAR-GROUP/hpx/issues/916
[3978] https://github.com/STEllAR-GROUP/hpx/issues/915
[3979] https://github.com/STEllAR-GROUP/hpx/issues/914
[3980] https://github.com/STEllAR-GROUP/hpx/issues/913
[3981] https://github.com/STEllAR-GROUP/hpx/issues/911
[3982] https://github.com/STEllAR-GROUP/hpx/issues/910
[3983] https://github.com/STEllAR-GROUP/hpx/issues/909
[3984] https://github.com/STEllAR-GROUP/hpx/issues/908
[3985] https://github.com/STEllAR-GROUP/hpx/issues/906
[3986] https://github.com/STEllAR-GROUP/hpx/issues/905
[3987] https://github.com/STEllAR-GROUP/hpx/issues/904
[3988] https://github.com/STEllAR-GROUP/hpx/issues/903

- Issue #902[3989] - Add `hpx::lcos::reduce` collective operation
- Issue #901[3990] - Web documentation is not searchable
- Issue #900[3991] - Web documentation for trunk has no index
- Issue #898[3992] - Some tests fail with GCC 4.8.1 and MPI parcel port
- Issue #897[3993] - HWLOC causes failures on Mac
- Issue #896[3994] - pu-offset leads to startup error
- Issue #895[3995] - `hpx::get_locality_name` not defined
- Issue #894[3996] - Race condition at shutdown
- Issue #893[3997] - `--hpx:print-bind` switches std::cout to hexadecimal mode
- Issue #892[3998] - `hwloc_topology_load` can be expensive – don't call multiple times
- Issue #891[3999] - The documentation for `get_locality_name` is wrong
- Issue #890[4000] - `--hpx:print-bind` should not exit
- Issue #889[4001] - `--hpx:debug-hpx-log=FILE` does not work
- Issue #888[4002] - MPI parcelport does not exit cleanly for –hpx:print-bind
- Issue #887[4003] - Choose thread affinities more cleverly
- Issue #886[4004] - Logging documentation is confusing
- Issue #885[4005] - Two threads are slower than one
- Issue #884[4006] - is_callable failing with member pointers in C++11
- Issue #883[4007] - Need help with is_callable_test
- Issue #882[4008] - tests.regressions.lcos.future_hang_on_get does not terminate
- Issue #881[4009] - tests/regressions/block_matrix/matrix.hh won't compile with GCC 4.8.1
- Issue #880[4010] - HPX does not work on OS X
- Issue #878[4011] - `future::unwrap` triggers assertion

[3989] https://github.com/STEllAR-GROUP/hpx/issues/902
[3990] https://github.com/STEllAR-GROUP/hpx/issues/901
[3991] https://github.com/STEllAR-GROUP/hpx/issues/900
[3992] https://github.com/STEllAR-GROUP/hpx/issues/898
[3993] https://github.com/STEllAR-GROUP/hpx/issues/897
[3994] https://github.com/STEllAR-GROUP/hpx/issues/896
[3995] https://github.com/STEllAR-GROUP/hpx/issues/895
[3996] https://github.com/STEllAR-GROUP/hpx/issues/894
[3997] https://github.com/STEllAR-GROUP/hpx/issues/893
[3998] https://github.com/STEllAR-GROUP/hpx/issues/892
[3999] https://github.com/STEllAR-GROUP/hpx/issues/891
[4000] https://github.com/STEllAR-GROUP/hpx/issues/890
[4001] https://github.com/STEllAR-GROUP/hpx/issues/889
[4002] https://github.com/STEllAR-GROUP/hpx/issues/888
[4003] https://github.com/STEllAR-GROUP/hpx/issues/887
[4004] https://github.com/STEllAR-GROUP/hpx/issues/886
[4005] https://github.com/STEllAR-GROUP/hpx/issues/885
[4006] https://github.com/STEllAR-GROUP/hpx/issues/884
[4007] https://github.com/STEllAR-GROUP/hpx/issues/883
[4008] https://github.com/STEllAR-GROUP/hpx/issues/882
[4009] https://github.com/STEllAR-GROUP/hpx/issues/881
[4010] https://github.com/STEllAR-GROUP/hpx/issues/880
[4011] https://github.com/STEllAR-GROUP/hpx/issues/878

- Issue #877[4012] - "make tests" has build errors on Ubuntu 12.10
- Issue #876[4013] - tcmalloc is used by default, even if it is not present
- Issue #875[4014] - global_fixture is defined in a header file
- Issue #874[4015] - Some tests take very long
- Issue #873[4016] - Add block-matrix code as regression test
- Issue #872[4017] - HPX documentation does not say how to run tests with detailed output
- Issue #871[4018] - All tests fail with "make test"
- Issue #870[4019] - Please explicitly disable serialization in classes that don't support it
- Issue #868[4020] - boost_any test failing
- Issue #867[4021] - Reduce the number of copies of `hpx::function` arguments
- Issue #863[4022] - Futures should not require a default constructor
- Issue #862[4023] - value_or_error shall not default construct its result
- Issue #861[4024] - `HPX_UNUSED` macro
- Issue #860[4025] - Add functionality to copy construct a component
- Issue #859[4026] - `hpx::endl` should flush
- Issue #858[4027] - Create `hpx::get_ptr<>` allowing to access component implementation
- Issue #855[4028] - Implement `hpx::INVOKE`
- Issue #854[4029] - `hpx/hpx.hpp` does not include `hpx/include/iostreams.hpp`
- Issue #853[4030] - Feature request: null future
- Issue #852[4031] - Feature request: Locality names
- Issue #851[4032] - `hpx::cout` output does not appear on screen
- Issue #849[4033] - All tests fail on OS X after installing
- Issue #848[4034] - Update OS X build instructions

4012 https://github.com/STEllAR-GROUP/hpx/issues/877
4013 https://github.com/STEllAR-GROUP/hpx/issues/876
4014 https://github.com/STEllAR-GROUP/hpx/issues/875
4015 https://github.com/STEllAR-GROUP/hpx/issues/874
4016 https://github.com/STEllAR-GROUP/hpx/issues/873
4017 https://github.com/STEllAR-GROUP/hpx/issues/872
4018 https://github.com/STEllAR-GROUP/hpx/issues/871
4019 https://github.com/STEllAR-GROUP/hpx/issues/870
4020 https://github.com/STEllAR-GROUP/hpx/issues/868
4021 https://github.com/STEllAR-GROUP/hpx/issues/867
4022 https://github.com/STEllAR-GROUP/hpx/issues/863
4023 https://github.com/STEllAR-GROUP/hpx/issues/862
4024 https://github.com/STEllAR-GROUP/hpx/issues/861
4025 https://github.com/STEllAR-GROUP/hpx/issues/860
4026 https://github.com/STEllAR-GROUP/hpx/issues/859
4027 https://github.com/STEllAR-GROUP/hpx/issues/858
4028 https://github.com/STEllAR-GROUP/hpx/issues/855
4029 https://github.com/STEllAR-GROUP/hpx/issues/854
4030 https://github.com/STEllAR-GROUP/hpx/issues/853
4031 https://github.com/STEllAR-GROUP/hpx/issues/852
4032 https://github.com/STEllAR-GROUP/hpx/issues/851
4033 https://github.com/STEllAR-GROUP/hpx/issues/849
4034 https://github.com/STEllAR-GROUP/hpx/issues/848

- Issue #846[4035] - Update hpx_external_example
- Issue #845[4036] - Issues with having both debug and release modules in the same directory
- Issue #844[4037] - Create configuration header
- Issue #843[4038] - Tests should use CTest
- Issue #842[4039] - Remove buffer_pool from MPI parcelport
- Issue #841[4040] - Add possibility to broadcast an index with hpx::lcos::broadcast
- Issue #838[4041] - Simplify `util::tuple`
- Issue #837[4042] - Adopt boost::tuple tests for `util::tuple`
- Issue #836[4043] - Adopt boost::function tests for `util::function`
- Issue #835[4044] - Tuple interface missing pieces
- Issue #833[4045] - Partially preprocessing files not working
- Issue #832[4046] - Native papi counters do not work with wild cards
- Issue #831[4047] - Arithmetics counter fails if only one parameter is given
- Issue #830[4048] - Convert hpx::util::function to use new scheme for serializing its base pointer
- Issue #829[4049] - Consistently use `decay<T>` instead of `remove_const< remove_reference<T>>`
- Issue #828[4050] - Update future implementation to N3721 and N3722
- Issue #827[4051] - Enable MPI parcelport for bootstrapping whenever application was started using mpirun
- Issue #826[4052] - Support command line option `--hpx:print-bind` even if `--hpx::bind` was not used
- Issue #825[4053] - Memory counters give segfault when attempting to use thread wild cards or numbers only total works
- Issue #824[4054] - Enable lambda functions to be used with hpx::async/hpx::apply
- Issue #823[4055] - Using a hashing filter
- Issue #822[4056] - Silence unused variable warning
- Issue #821[4057] - Detect if a function object is callable with given arguments

[4035] https://github.com/STEllAR-GROUP/hpx/issues/846
[4036] https://github.com/STEllAR-GROUP/hpx/issues/845
[4037] https://github.com/STEllAR-GROUP/hpx/issues/844
[4038] https://github.com/STEllAR-GROUP/hpx/issues/843
[4039] https://github.com/STEllAR-GROUP/hpx/issues/842
[4040] https://github.com/STEllAR-GROUP/hpx/issues/841
[4041] https://github.com/STEllAR-GROUP/hpx/issues/838
[4042] https://github.com/STEllAR-GROUP/hpx/issues/837
[4043] https://github.com/STEllAR-GROUP/hpx/issues/836
[4044] https://github.com/STEllAR-GROUP/hpx/issues/835
[4045] https://github.com/STEllAR-GROUP/hpx/issues/833
[4046] https://github.com/STEllAR-GROUP/hpx/issues/832
[4047] https://github.com/STEllAR-GROUP/hpx/issues/831
[4048] https://github.com/STEllAR-GROUP/hpx/issues/830
[4049] https://github.com/STEllAR-GROUP/hpx/issues/829
[4050] https://github.com/STEllAR-GROUP/hpx/issues/828
[4051] https://github.com/STEllAR-GROUP/hpx/issues/827
[4052] https://github.com/STEllAR-GROUP/hpx/issues/826
[4053] https://github.com/STEllAR-GROUP/hpx/issues/825
[4054] https://github.com/STEllAR-GROUP/hpx/issues/824
[4055] https://github.com/STEllAR-GROUP/hpx/issues/823
[4056] https://github.com/STEllAR-GROUP/hpx/issues/822
[4057] https://github.com/STEllAR-GROUP/hpx/issues/821

- Issue #820[4058] - Allow wildcards to be used for performance counter names
- Issue #819[4059] - Make the AGAS symbolic name registry distributed
- Issue #818[4060] - Add future::then() overload taking an executor
- Issue #817[4061] - Fixed typo
- Issue #815[4062] - Create an lco that is performing an efficient broadcast of actions
- Issue #814[4063] - Papi counters cannot specify thread#* to get the counts for all threads
- Issue #813[4064] - Scoped unlock
- Issue #811[4065] - simple_central_tuplespace_client run error
- Issue #810[4066] - ostream error when << any objects
- Issue #809[4067] - Optimize parcel serialization
- Issue #808[4068] - HPX applications throw exception when executed from the build directory
- Issue #807[4069] - Create performance counters exposing overall AGAS statistics
- Issue #795[4070] - Create timed make_ready_future
- Issue #794[4071] - Create heterogeneous `when_all`/`when_any`/etc.
- Issue #721[4072] - Make HPX usable for Xeon Phi
- Issue #694[4073] - CMake should complain if you attempt to build an example without its dependencies
- Issue #692[4074] - SLURM support broken
- Issue #683[4075] - python/hpx/process.py imports epoll on all platforms
- Issue #619[4076] - Automate the doc building process
- Issue #600[4077] - GTC performance broken
- Issue #577[4078] - Allow for zero copy serialization/networking
- Issue #551[4079] - Change executable names to have debug postfix in Debug builds
- Issue #544[4080] - Write a custom .lib file on Windows pulling in hpx_init and hpx.dll, phase out hpx_init

4058 https://github.com/STEllAR-GROUP/hpx/issues/820
4059 https://github.com/STEllAR-GROUP/hpx/issues/819
4060 https://github.com/STEllAR-GROUP/hpx/issues/818
4061 https://github.com/STEllAR-GROUP/hpx/issues/817
4062 https://github.com/STEllAR-GROUP/hpx/issues/815
4063 https://github.com/STEllAR-GROUP/hpx/issues/814
4064 https://github.com/STEllAR-GROUP/hpx/issues/813
4065 https://github.com/STEllAR-GROUP/hpx/issues/811
4066 https://github.com/STEllAR-GROUP/hpx/issues/810
4067 https://github.com/STEllAR-GROUP/hpx/issues/809
4068 https://github.com/STEllAR-GROUP/hpx/issues/808
4069 https://github.com/STEllAR-GROUP/hpx/issues/807
4070 https://github.com/STEllAR-GROUP/hpx/issues/795
4071 https://github.com/STEllAR-GROUP/hpx/issues/794
4072 https://github.com/STEllAR-GROUP/hpx/issues/721
4073 https://github.com/STEllAR-GROUP/hpx/issues/694
4074 https://github.com/STEllAR-GROUP/hpx/issues/692
4075 https://github.com/STEllAR-GROUP/hpx/issues/683
4076 https://github.com/STEllAR-GROUP/hpx/issues/619
4077 https://github.com/STEllAR-GROUP/hpx/issues/600
4078 https://github.com/STEllAR-GROUP/hpx/issues/577
4079 https://github.com/STEllAR-GROUP/hpx/issues/551
4080 https://github.com/STEllAR-GROUP/hpx/issues/544

- Issue #534[4081] - `hpx::init` should take functions by `std::function` and should accept all forms of hpx_main
- Issue #508[4082] - FindPackage fails to set FOO_LIBRARY_DIR
- Issue #506[4083] - Add cmake support to generate ini files for external applications
- Issue #470[4084] - Changing build-type after configure does not update boost library names
- Issue #453[4085] - Document `hpx_run_tests.py`
- Issue #445[4086] - Significant performance mismatch between MPI and HPX in SMP for allgather example
- Issue #443[4087] - Make docs viewable from build directory
- Issue #421[4088] - Support multiple HPX instances per node in a batch environment like PBS or SLURM
- Issue #316[4089] - Add message size limitation
- Issue #249[4090] - Clean up locking code in big boot barrier
- Issue #136[4091] - Persistent CMake variables need to be marked as cache variables

2.11.16 *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[4092] and related proposals to the C++ standardization committee (such as N3632[4093] and N3857[4094]).
- 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.

[4081] https://github.com/STEllAR-GROUP/hpx/issues/534
[4082] https://github.com/STEllAR-GROUP/hpx/issues/508
[4083] https://github.com/STEllAR-GROUP/hpx/issues/506
[4084] https://github.com/STEllAR-GROUP/hpx/issues/470
[4085] https://github.com/STEllAR-GROUP/hpx/issues/453
[4086] https://github.com/STEllAR-GROUP/hpx/issues/445
[4087] https://github.com/STEllAR-GROUP/hpx/issues/443
[4088] https://github.com/STEllAR-GROUP/hpx/issues/421
[4089] https://github.com/STEllAR-GROUP/hpx/issues/316
[4090] https://github.com/STEllAR-GROUP/hpx/issues/249
[4091] https://github.com/STEllAR-GROUP/hpx/issues/136
[4092] http://www.open-std.org/jtc1/sc22/wg21
[4093] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3632.html
[4094] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf

- 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[4095]. 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[4096]).

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[4097] - make (all) in examples folder does nothing
- Issue #805[4098] - Adding the introduction and fixing DOCBOOK dependencies for Windows use
- Issue #804[4099] - Add stackless (non-suspendable) thread type
- Issue #803[4100] - Create proper serialization support functions for util::tuple
- Issue #800[4101] - Add possibility to disable array optimizations during serialization
- Issue #798[4102] - HPX_LIMIT does not work for local dataflow
- Issue #797[4103] - Create a parcelport which uses MPI
- Issue #796[4104] - Problem with Large Numbers of Threads
- Issue #793[4105] - Changing dataflow test case to hang consistently
- Issue #792[4106] - CMake Error
- Issue #791[4107] - Problems with local::dataflow
- Issue #790[4108] - wait_for() doesn't compile
- Issue #789[4109] - HPX with Intel compiler segfaults

[4095] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf
[4096] https://en.wikipedia.org/wiki/Linda_(coordination_language)
[4097] https://github.com/STEllAR-GROUP/hpx/issues/806
[4098] https://github.com/STEllAR-GROUP/hpx/issues/805
[4099] https://github.com/STEllAR-GROUP/hpx/issues/804
[4100] https://github.com/STEllAR-GROUP/hpx/issues/803
[4101] https://github.com/STEllAR-GROUP/hpx/issues/800
[4102] https://github.com/STEllAR-GROUP/hpx/issues/798
[4103] https://github.com/STEllAR-GROUP/hpx/issues/797
[4104] https://github.com/STEllAR-GROUP/hpx/issues/796
[4105] https://github.com/STEllAR-GROUP/hpx/issues/793
[4106] https://github.com/STEllAR-GROUP/hpx/issues/792
[4107] https://github.com/STEllAR-GROUP/hpx/issues/791
[4108] https://github.com/STEllAR-GROUP/hpx/issues/790
[4109] https://github.com/STEllAR-GROUP/hpx/issues/789

- Issue #788[4110] - Intel compiler support
- Issue #787[4111] - Fixed SFINAEd specializations
- Issue #786[4112] - Memory issues during benchmarking.
- Issue #785[4113] - Create an API allowing to register external threads with HPX
- Issue #784[4114] - util::plugin is throwing an error when a symbol is not found
- Issue #783[4115] - How does hpx:bind work?
- Issue #782[4116] - Added quotes around STRING REPLACE potentially empty arguments
- Issue #781[4117] - Make sure no exceptions propagate into the thread manager
- Issue #780[4118] - Allow arithmetics performance counters to expand its parameters
- Issue #779[4119] - Test case for 778
- Issue #778[4120] - Swapping futures segfaults
- Issue #777[4121] - hpx::lcos::details::when_xxx don't restore completion handlers
- Issue #776[4122] - Compiler chokes on dataflow overload with launch policy
- Issue #775[4123] - Runtime error with local dataflow (copying futures?)
- Issue #774[4124] - Using local dataflow without explicit namespace
- Issue #773[4125] - Local dataflow with unwrap: functor operators need to be const
- Issue #772[4126] - Allow (remote) actions to return a future
- Issue #771[4127] - Setting HPX_LIMIT gives huge boost MPL errors
- Issue #770[4128] - Add launch policy to (local) dataflow
- Issue #769[4129] - Make compile time configuration information available
- Issue #768[4130] - Const correctness problem in local dataflow
- Issue #767[4131] - Add launch policies to async
- Issue #766[4132] - Mark data structures for optimized (array based) serialization

[4110] https://github.com/STEllAR-GROUP/hpx/issues/788
[4111] https://github.com/STEllAR-GROUP/hpx/issues/787
[4112] https://github.com/STEllAR-GROUP/hpx/issues/786
[4113] https://github.com/STEllAR-GROUP/hpx/issues/785
[4114] https://github.com/STEllAR-GROUP/hpx/issues/784
[4115] https://github.com/STEllAR-GROUP/hpx/issues/783
[4116] https://github.com/STEllAR-GROUP/hpx/issues/782
[4117] https://github.com/STEllAR-GROUP/hpx/issues/781
[4118] https://github.com/STEllAR-GROUP/hpx/issues/780
[4119] https://github.com/STEllAR-GROUP/hpx/issues/779
[4120] https://github.com/STEllAR-GROUP/hpx/issues/778
[4121] https://github.com/STEllAR-GROUP/hpx/issues/777
[4122] https://github.com/STEllAR-GROUP/hpx/issues/776
[4123] https://github.com/STEllAR-GROUP/hpx/issues/775
[4124] https://github.com/STEllAR-GROUP/hpx/issues/774
[4125] https://github.com/STEllAR-GROUP/hpx/issues/773
[4126] https://github.com/STEllAR-GROUP/hpx/issues/772
[4127] https://github.com/STEllAR-GROUP/hpx/issues/771
[4128] https://github.com/STEllAR-GROUP/hpx/issues/770
[4129] https://github.com/STEllAR-GROUP/hpx/issues/769
[4130] https://github.com/STEllAR-GROUP/hpx/issues/768
[4131] https://github.com/STEllAR-GROUP/hpx/issues/767
[4132] https://github.com/STEllAR-GROUP/hpx/issues/766

- Issue #765[4133] - Align hpx::any with N3508: Any Library Proposal (Revision 2)
- Issue #764[4134] - Align hpx::future with newest N3558: A Standardized Representation of Asynchronous Operations
- Issue #762[4135] - added a human readable output for the ping pong example
- Issue #761[4136] - Ambiguous typename when constructing derived component
- Issue #760[4137] - Simple components can not be derived
- Issue #759[4138] - make install doesn't give a complete install
- Issue #758[4139] - Stack overflow when using locking_hook<>
- Issue #757[4140] - copy paste error; unsupported function overloading
- Issue #756[4141] - GTCX runtime issue in Gordon
- Issue #755[4142] - Papi counters don't work with reset and evaluate API's
- Issue #753[4143] - cmake bugfix and improved component action docs
- Issue #752[4144] - hpx simple component docs
- Issue #750[4145] - Add hpx::util::any
- Issue #749[4146] - Thread phase counter is not reset
- Issue #748[4147] - Memory performance counter are not registered
- Issue #747[4148] - Create performance counters exposing arithmetic operations
- Issue #745[4149] - apply_callback needs to invoke callback when applied locally
- Issue #744[4150] - CMake fixes
- Issue #743[4151] - Problem Building github version of HPX
- Issue #742[4152] - Remove HPX_STD_BIND
- Issue #741[4153] - assertion 'px != 0' failed: HPX(assertion_failure) for low numbers of OS threads
- Issue #739[4154] - Performance counters do not count to the end of the program or evalution
- Issue #738[4155] - Dedicated AGAS server runs don't work; console ignores -a option.

4133 https://github.com/STEllAR-GROUP/hpx/issues/765
4134 https://github.com/STEllAR-GROUP/hpx/issues/764
4135 https://github.com/STEllAR-GROUP/hpx/issues/762
4136 https://github.com/STEllAR-GROUP/hpx/issues/761
4137 https://github.com/STEllAR-GROUP/hpx/issues/760
4138 https://github.com/STEllAR-GROUP/hpx/issues/759
4139 https://github.com/STEllAR-GROUP/hpx/issues/758
4140 https://github.com/STEllAR-GROUP/hpx/issues/757
4141 https://github.com/STEllAR-GROUP/hpx/issues/756
4142 https://github.com/STEllAR-GROUP/hpx/issues/755
4143 https://github.com/STEllAR-GROUP/hpx/issues/753
4144 https://github.com/STEllAR-GROUP/hpx/issues/752
4145 https://github.com/STEllAR-GROUP/hpx/issues/750
4146 https://github.com/STEllAR-GROUP/hpx/issues/749
4147 https://github.com/STEllAR-GROUP/hpx/issues/748
4148 https://github.com/STEllAR-GROUP/hpx/issues/747
4149 https://github.com/STEllAR-GROUP/hpx/issues/745
4150 https://github.com/STEllAR-GROUP/hpx/issues/744
4151 https://github.com/STEllAR-GROUP/hpx/issues/743
4152 https://github.com/STEllAR-GROUP/hpx/issues/742
4153 https://github.com/STEllAR-GROUP/hpx/issues/741
4154 https://github.com/STEllAR-GROUP/hpx/issues/739
4155 https://github.com/STEllAR-GROUP/hpx/issues/738

- Issue #737[4156] - Missing bind overloads
- Issue #736[4157] - Performance counter wildcards do not always work
- Issue #735[4158] - Create native ibverbs parcelport based on rdma operations
- Issue #734[4159] - Threads stolen performance counter total is incorrect
- Issue #733[4160] - Test benchmarks need to be checked and fixed
- Issue #732[4161] - Build fails with Mac, using mac ports clang-3.3 on latest git branch
- Issue #731[4162] - Add global start/stop API for performance counters
- Issue #730[4163] - Performance counter values are apparently incorrect
- Issue #729[4164] - Unhandled switch
- Issue #728[4165] - Serialization of hpx::util::function between two localities causes seg faults
- Issue #727[4166] - Memory counters on Mac OS X
- Issue #725[4167] - Restore original thread priority on resume
- Issue #724[4168] - Performance benchmarks do not depend on main HPX libraries
- Issue #723[4169] - [teletype]–hpx:nodes=``cat $PBS_NODEFILE`` works; –hpx:nodefile=$PBS_NODEFILE does not.[c++]
- Issue #722[4170] - Fix binding const member functions as actions
- Issue #719[4171] - Create performance counter exposing compression ratio
- Issue #718[4172] - Add possibility to compress parcel data
- Issue #717[4173] - strip_credit_from_gid has misleading semantics
- Issue #716[4174] - Non-option arguments to programs run using `pbsdsh` must be before `--hpx:nodes`, contrary to directions
- Issue #715[4175] - Re-thrown exceptions should retain the original call site
- Issue #714[4176] - failed assertion in debug mode
- Issue #713[4177] - Add performance counters monitoring connection caches

4156 https://github.com/STEllAR-GROUP/hpx/issues/737
4157 https://github.com/STEllAR-GROUP/hpx/issues/736
4158 https://github.com/STEllAR-GROUP/hpx/issues/735
4159 https://github.com/STEllAR-GROUP/hpx/issues/734
4160 https://github.com/STEllAR-GROUP/hpx/issues/733
4161 https://github.com/STEllAR-GROUP/hpx/issues/732
4162 https://github.com/STEllAR-GROUP/hpx/issues/731
4163 https://github.com/STEllAR-GROUP/hpx/issues/730
4164 https://github.com/STEllAR-GROUP/hpx/issues/729
4165 https://github.com/STEllAR-GROUP/hpx/issues/728
4166 https://github.com/STEllAR-GROUP/hpx/issues/727
4167 https://github.com/STEllAR-GROUP/hpx/issues/725
4168 https://github.com/STEllAR-GROUP/hpx/issues/724
4169 https://github.com/STEllAR-GROUP/hpx/issues/723
4170 https://github.com/STEllAR-GROUP/hpx/issues/722
4171 https://github.com/STEllAR-GROUP/hpx/issues/719
4172 https://github.com/STEllAR-GROUP/hpx/issues/718
4173 https://github.com/STEllAR-GROUP/hpx/issues/717
4174 https://github.com/STEllAR-GROUP/hpx/issues/716
4175 https://github.com/STEllAR-GROUP/hpx/issues/715
4176 https://github.com/STEllAR-GROUP/hpx/issues/714
4177 https://github.com/STEllAR-GROUP/hpx/issues/713

- Issue #712[4178] - Adjust parcel related performance counters to be connection type specific
- Issue #711[4179] - configuration failure
- Issue #710[4180] - Error "timed out while trying to find room in the connection cache" when trying to start multiple localities on a single computer
- Issue #709[4181] - Add new thread state 'staged' referring to task descriptions
- Issue #708[4182] - Detect/mitigate bad non-system installs of GCC on Redhat systems
- Issue #707[4183] - Many examples do not link with Git HEAD version
- Issue #706[4184] - `hpx::init` removes portions of non-option command line arguments before last = sign
- Issue #705[4185] - Create rolling average and median aggregating performance counters
- Issue #704[4186] - Create performance counter to expose thread queue waiting time
- Issue #703[4187] - Add support to HPX build system to find librcrtool.a and related headers
- Issue #699[4188] - Generalize instrumentation support
- Issue #698[4189] - compilation failure with hwloc absent
- Issue #697[4190] - Performance counter counts should be zero indexed
- Issue #696[4191] - Distributed problem
- Issue #695[4192] - Bad perf counter time printed
- Issue #693[4193] - `--help` doesn't print component specific command line options
- Issue #692[4194] - SLURM support broken
- Issue #691[4195] - exception while executing any application linked with hwloc
- Issue #690[4196] - thread_id_test and thread_launcher_test failing
- Issue #689[4197] - Make the buildbots use hwloc
- Issue #687[4198] - compilation error fix (hwloc_topology)
- Issue #686[4199] - Linker Error for Applications
- Issue #684[4200] - Pinning of service thread fails when number of worker threads equals the number of cores

[4178] https://github.com/STEllAR-GROUP/hpx/issues/712
[4179] https://github.com/STEllAR-GROUP/hpx/issues/711
[4180] https://github.com/STEllAR-GROUP/hpx/issues/710
[4181] https://github.com/STEllAR-GROUP/hpx/issues/709
[4182] https://github.com/STEllAR-GROUP/hpx/issues/708
[4183] https://github.com/STEllAR-GROUP/hpx/issues/707
[4184] https://github.com/STEllAR-GROUP/hpx/issues/706
[4185] https://github.com/STEllAR-GROUP/hpx/issues/705
[4186] https://github.com/STEllAR-GROUP/hpx/issues/704
[4187] https://github.com/STEllAR-GROUP/hpx/issues/703
[4188] https://github.com/STEllAR-GROUP/hpx/issues/699
[4189] https://github.com/STEllAR-GROUP/hpx/issues/698
[4190] https://github.com/STEllAR-GROUP/hpx/issues/697
[4191] https://github.com/STEllAR-GROUP/hpx/issues/696
[4192] https://github.com/STEllAR-GROUP/hpx/issues/695
[4193] https://github.com/STEllAR-GROUP/hpx/issues/693
[4194] https://github.com/STEllAR-GROUP/hpx/issues/692
[4195] https://github.com/STEllAR-GROUP/hpx/issues/691
[4196] https://github.com/STEllAR-GROUP/hpx/issues/690
[4197] https://github.com/STEllAR-GROUP/hpx/issues/689
[4198] https://github.com/STEllAR-GROUP/hpx/issues/687
[4199] https://github.com/STEllAR-GROUP/hpx/issues/686
[4200] https://github.com/STEllAR-GROUP/hpx/issues/684

- Issue #682[4201] - Add performance counters exposing number of stolen threads
- Issue #681[4202] - Add apply_continue for asynchronous chaining of actions
- Issue #679[4203] - Remove obsolete async_callback API functions
- Issue #678[4204] - Add new API for setting/triggering LCOs
- Issue #677[4205] - Add async_continue for true continuation style actions
- Issue #676[4206] - Buildbot for gcc 4.4 broken
- Issue #675[4207] - Partial preprocessing broken
- Issue #674[4208] - HPX segfaults when built with gcc 4.7
- Issue #673[4209] - `use_guard_pages` has inconsistent preprocessor guards
- Issue #672[4210] - External build breaks if library path has spaces
- Issue #671[4211] - release tarballs are tarbombs
- Issue #670[4212] - CMake won't find Boost headers in layout=versioned install
- Issue #669[4213] - Links in docs to source files broken if not installed
- Issue #667[4214] - Not reading ini file properly
- Issue #664[4215] - Adapt new meanings of 'const' and 'mutable'
- Issue #661[4216] - Implement BTL Parcel port
- Issue #655[4217] - Make HPX work with the "decltype" result_of
- Issue #647[4218] - documentation for specifying the number of high priority threads `--hpx:high-priority-threads`
- Issue #643[4219] - Error parsing host file
- Issue #642[4220] - HWLoc issue with TAU
- Issue #639[4221] - Logging potentially suspends a running thread
- Issue #634[4222] - Improve error reporting from parcel layer
- Issue #627[4223] - Add tests for async and apply overloads that accept regular C++ functions

4201 https://github.com/STEllAR-GROUP/hpx/issues/682
4202 https://github.com/STEllAR-GROUP/hpx/issues/681
4203 https://github.com/STEllAR-GROUP/hpx/issues/679
4204 https://github.com/STEllAR-GROUP/hpx/issues/678
4205 https://github.com/STEllAR-GROUP/hpx/issues/677
4206 https://github.com/STEllAR-GROUP/hpx/issues/676
4207 https://github.com/STEllAR-GROUP/hpx/issues/675
4208 https://github.com/STEllAR-GROUP/hpx/issues/674
4209 https://github.com/STEllAR-GROUP/hpx/issues/673
4210 https://github.com/STEllAR-GROUP/hpx/issues/672
4211 https://github.com/STEllAR-GROUP/hpx/issues/671
4212 https://github.com/STEllAR-GROUP/hpx/issues/670
4213 https://github.com/STEllAR-GROUP/hpx/issues/669
4214 https://github.com/STEllAR-GROUP/hpx/issues/667
4215 https://github.com/STEllAR-GROUP/hpx/issues/664
4216 https://github.com/STEllAR-GROUP/hpx/issues/661
4217 https://github.com/STEllAR-GROUP/hpx/issues/655
4218 https://github.com/STEllAR-GROUP/hpx/issues/647
4219 https://github.com/STEllAR-GROUP/hpx/issues/643
4220 https://github.com/STEllAR-GROUP/hpx/issues/642
4221 https://github.com/STEllAR-GROUP/hpx/issues/639
4222 https://github.com/STEllAR-GROUP/hpx/issues/634
4223 https://github.com/STEllAR-GROUP/hpx/issues/627

- Issue #626[4224] - hpx/future.hpp header
- Issue #601[4225] - Intel support
- Issue #557[4226] - Remove action codes
- Issue #531[4227] - AGAS request and response classes should use switch statements
- Issue #529[4228] - Investigate the state of hwloc support
- Issue #526[4229] - Make HPX aware of hyper-threading
- Issue #518[4230] - Create facilities allowing to use plain arrays as action arguments
- Issue #473[4231] - hwloc thread binding is broken on CPUs with hyperthreading
- Issue #383[4232] - Change result type detection for hpx::util::bind to use result_of protocol
- Issue #341[4233] - Consolidate route code
- Issue #219[4234] - Only copy arguments into actions once
- Issue #177[4235] - Implement distributed AGAS
- Issue #43[4236] - Support for Darwin (Xcode + Clang)

2.11.17 *HPX* V0.9.5 (Jan 16, 2013)

We have had over 1000 commits since the last release and we have closed roughly 150 tickets (bugs, feature requests, etc.).

General changes

This release is continuing along the lines of code and API consolidation, and overall usability inprovements. We dedicated much attention to performance and we were able to significantly improve the threading and networking subsystems.

We successfully ported *HPX* to the Android platform. *HPX* applications now not only can run on mobile devices, but we support heterogeneous applications running across architecture boundaries. At the Supercomputing Conference 2012 we demonstrated connecting Android tablets to simulations running on a Linux cluster. The Android tablet was used to query performance counters from the Linux simulation and to steer its parameters.

We successfully ported *HPX* to Mac OSX (using the Clang compiler). Thanks to Pyry Jahkola for contributing the corresponding patches. Please see the section *How to install HPX on OS X (Mac)* 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

[4224] https://github.com/STEllAR-GROUP/hpx/issues/626
[4225] https://github.com/STEllAR-GROUP/hpx/issues/601
[4226] https://github.com/STEllAR-GROUP/hpx/issues/557
[4227] https://github.com/STEllAR-GROUP/hpx/issues/531
[4228] https://github.com/STEllAR-GROUP/hpx/issues/529
[4229] https://github.com/STEllAR-GROUP/hpx/issues/526
[4230] https://github.com/STEllAR-GROUP/hpx/issues/518
[4231] https://github.com/STEllAR-GROUP/hpx/issues/473
[4232] https://github.com/STEllAR-GROUP/hpx/issues/383
[4233] https://github.com/STEllAR-GROUP/hpx/issues/341
[4234] https://github.com/STEllAR-GROUP/hpx/issues/219
[4235] https://github.com/STEllAR-GROUP/hpx/issues/177
[4236] https://github.com/STEllAR-GROUP/hpx/issues/43

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[4237] 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[4238]).

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[4239] - Segfault on calling hpx::finalize twice
- Issue #665[4240] - Adding declaration num_of_cores
- Issue #662[4241] - pkgconfig is building wrong
- Issue #660[4242] - Need uninterrupt function
- Issue #659[4243] - Move our logging library into a different namespace
- Issue #658[4244] - Dynamic performance counter types are broken
- Issue #657[4245] - HPX v0.9.5 (RC1) hello_world example segfaulting
- Issue #656[4246] - Define the affinity of parcel-pool, io-pool, and timer-pool threads
- Issue #654[4247] - Integrate the Boost auto_index tool with documentation
- Issue #653[4248] - Make HPX build on OS X + Clang + libc++
- Issue #651[4249] - Add fine-grained control for thread pinning
- Issue #650[4250] - Command line no error message when using -hpx:(anything)
- Issue #645[4251] - Command line aliases don't work in [teletype]``@file``[c++]
- Issue #644[4252] - Terminated threads are not always properly cleaned up

[4237] https://github.com/STEllAR-GROUP/hpx/
[4238] https://www.boost.org/LICENSE_1_0.txt
[4239] https://github.com/STEllAR-GROUP/hpx/issues/666
[4240] https://github.com/STEllAR-GROUP/hpx/issues/665
[4241] https://github.com/STEllAR-GROUP/hpx/issues/662
[4242] https://github.com/STEllAR-GROUP/hpx/issues/660
[4243] https://github.com/STEllAR-GROUP/hpx/issues/659
[4244] https://github.com/STEllAR-GROUP/hpx/issues/658
[4245] https://github.com/STEllAR-GROUP/hpx/issues/657
[4246] https://github.com/STEllAR-GROUP/hpx/issues/656
[4247] https://github.com/STEllAR-GROUP/hpx/issues/654
[4248] https://github.com/STEllAR-GROUP/hpx/issues/653
[4249] https://github.com/STEllAR-GROUP/hpx/issues/651
[4250] https://github.com/STEllAR-GROUP/hpx/issues/650
[4251] https://github.com/STEllAR-GROUP/hpx/issues/645
[4252] https://github.com/STEllAR-GROUP/hpx/issues/644

- Issue #640[4253] - `future_data<T>::set_on_completed_` used without locks
- Issue #638[4254] - hpx build with intel compilers fails on linux
- Issue #637[4255] - –copy-dt-needed-entries breaks with gold
- Issue #635[4256] - Boost V1.53 will add Boost.Lockfree and Boost.Atomic
- Issue #633[4257] - Re-add examples to final 0.9.5 release
- Issue #632[4258] - Example `thread_aware_timer` is broken
- Issue #631[4259] - FFT application throws error in parcellayer
- Issue #630[4260] - Event synchronization example is broken
- Issue #629[4261] - Waiting on futures hangs
- Issue #628[4262] - Add an `HPX_ALWAYS_ASSERT` macro
- Issue #625[4263] - Port coroutines context switch benchmark
- Issue #621[4264] - New INI section for stack sizes
- Issue #618[4265] - pkg_config support does not work with a HPX debug build
- Issue #617[4266] - hpx/external/logging/boost/logging/detail/cache_before_init.hpp:139:67: error: 'get_thread_id' was not declared in this scope
- Issue #616[4267] - Change wait_xxx not to use locking
- Issue #615[4268] - Revert visibility 'fix' (fb0b6b8245dad1127b0c25ebafd9386b3945cca9)
- Issue #614[4269] - Fix Dataflow linker error
- Issue #613[4270] - find_here should throw an exception on failure
- Issue #612[4271] - Thread phase doesn't show up in debug mode
- Issue #611[4272] - Make stack guard pages configurable at runtime (initialization time)
- Issue #610[4273] - Co-Locate Components
- Issue #609[4274] - future_overhead
- Issue #608[4275] - `--hpx:list-counter-infos` problem

4253 https://github.com/STEllAR-GROUP/hpx/issues/640
4254 https://github.com/STEllAR-GROUP/hpx/issues/638
4255 https://github.com/STEllAR-GROUP/hpx/issues/637
4256 https://github.com/STEllAR-GROUP/hpx/issues/635
4257 https://github.com/STEllAR-GROUP/hpx/issues/633
4258 https://github.com/STEllAR-GROUP/hpx/issues/632
4259 https://github.com/STEllAR-GROUP/hpx/issues/631
4260 https://github.com/STEllAR-GROUP/hpx/issues/630
4261 https://github.com/STEllAR-GROUP/hpx/issues/629
4262 https://github.com/STEllAR-GROUP/hpx/issues/628
4263 https://github.com/STEllAR-GROUP/hpx/issues/625
4264 https://github.com/STEllAR-GROUP/hpx/issues/621
4265 https://github.com/STEllAR-GROUP/hpx/issues/618
4266 https://github.com/STEllAR-GROUP/hpx/issues/617
4267 https://github.com/STEllAR-GROUP/hpx/issues/616
4268 https://github.com/STEllAR-GROUP/hpx/issues/615
4269 https://github.com/STEllAR-GROUP/hpx/issues/614
4270 https://github.com/STEllAR-GROUP/hpx/issues/613
4271 https://github.com/STEllAR-GROUP/hpx/issues/612
4272 https://github.com/STEllAR-GROUP/hpx/issues/611
4273 https://github.com/STEllAR-GROUP/hpx/issues/610
4274 https://github.com/STEllAR-GROUP/hpx/issues/609
4275 https://github.com/STEllAR-GROUP/hpx/issues/608

- Issue #607[4276] - Update Boost.Context based backend for coroutines
- Issue #606[4277] - 1d_wave_equation is not working
- Issue #605[4278] - Any C++ function that has serializable arguments and a serializable return type should be remotable
- Issue #604[4279] - Connecting localities isn't working anymore
- Issue #603[4280] - Do not verify any ini entries read from a file
- Issue #602[4281] - Rename argument_size to type_size/ added implementation to get parcel size
- Issue #599[4282] - Enable locality specific command line options
- Issue #598[4283] - Need an API that accesses the performance counter reporting the system uptime
- Issue #597[4284] - compiling on ranger
- Issue #595[4285] - I need a place to store data in a thread self pointer
- Issue #594[4286] - 32/64 interoperability
- Issue #593[4287] - Warn if logging is disabled at compile time but requested at runtime
- Issue #592[4288] - Add optional argument value to `--hpx:list-counters` and `--hpx:list-counter-infos`
- Issue #591[4289] - Allow for wildcards in performance counter names specified with `--hpx:print-counter`
- Issue #590[4290] - Local promise semantic differences
- Issue #589[4291] - Create API to query performance counter names
- Issue #587[4292] - Add get_num_localities and get_num_threads to AGAS API
- Issue #586[4293] - Adjust local AGAS cache size based on number of localities
- Issue #585[4294] - Error while using counters in HPX
- Issue #584[4295] - counting argument size of actions, initial pass.
- Issue #581[4296] - Remove `RemoteResult` template parameter for `future<>`
- Issue #580[4297] - Add possibility to hook into actions

[4276] https://github.com/STEllAR-GROUP/hpx/issues/607
[4277] https://github.com/STEllAR-GROUP/hpx/issues/606
[4278] https://github.com/STEllAR-GROUP/hpx/issues/605
[4279] https://github.com/STEllAR-GROUP/hpx/issues/604
[4280] https://github.com/STEllAR-GROUP/hpx/issues/603
[4281] https://github.com/STEllAR-GROUP/hpx/issues/602
[4282] https://github.com/STEllAR-GROUP/hpx/issues/599
[4283] https://github.com/STEllAR-GROUP/hpx/issues/598
[4284] https://github.com/STEllAR-GROUP/hpx/issues/597
[4285] https://github.com/STEllAR-GROUP/hpx/issues/595
[4286] https://github.com/STEllAR-GROUP/hpx/issues/594
[4287] https://github.com/STEllAR-GROUP/hpx/issues/593
[4288] https://github.com/STEllAR-GROUP/hpx/issues/592
[4289] https://github.com/STEllAR-GROUP/hpx/issues/591
[4290] https://github.com/STEllAR-GROUP/hpx/issues/590
[4291] https://github.com/STEllAR-GROUP/hpx/issues/589
[4292] https://github.com/STEllAR-GROUP/hpx/issues/587
[4293] https://github.com/STEllAR-GROUP/hpx/issues/586
[4294] https://github.com/STEllAR-GROUP/hpx/issues/585
[4295] https://github.com/STEllAR-GROUP/hpx/issues/584
[4296] https://github.com/STEllAR-GROUP/hpx/issues/581
[4297] https://github.com/STEllAR-GROUP/hpx/issues/580

- Issue #578[4298] - Use angle brackets in HPX error dumps
- Issue #576[4299] - Exception incorrectly thrown when `--help` is used
- Issue #575[4300] - HPX(bad_component_type) with gcc 4.7.2 and boost 1.51
- Issue #574[4301] - `--hpx:connect` command line parameter not working correctly
- Issue #571[4302] - `hpx::wait()` (callback version) should pass the future to the callback function
- Issue #570[4303] - `hpx::wait` should operate on `boost::arrays` and `std::lists`
- Issue #569[4304] - Add a logging sink for Android
- Issue #568[4305] - 2-argument version of `HPX_DEFINE_COMPONENT_ACTION`
- Issue #567[4306] - Connecting to a running HPX application works only once
- Issue #565[4307] - HPX doesn't shutdown properly
- Issue #564[4308] - Partial preprocessing of new component creation interface
- Issue #563[4309] - Add `hpx::start`/`hpx::stop` to avoid blocking main thread
- Issue #562[4310] - All command line arguments swallowed by hpx
- Issue #561[4311] - Boost.Tuple is not move aware
- Issue #558[4312] - `boost::shared_ptr<>` style semantics/syntax for client classes
- Issue #556[4313] - Creation of partially preprocessed headers should be enabled for Boost newer than V1.50
- Issue #555[4314] - `BOOST_FORCEINLINE` does not name a type
- Issue #554[4315] - Possible race condition in thread `get_id()`
- Issue #552[4316] - Move enable client_base
- Issue #550[4317] - Add stack size category 'huge'
- Issue #549[4318] - ShenEOS run seg-faults on single or distributed runs
- Issue #545[4319] - `AUTOGLOB` broken for add_hpx_component
- Issue #542[4320] - FindHPX_HDF5 still searches multiple times

4298 https://github.com/STEllAR-GROUP/hpx/issues/578
4299 https://github.com/STEllAR-GROUP/hpx/issues/576
4300 https://github.com/STEllAR-GROUP/hpx/issues/575
4301 https://github.com/STEllAR-GROUP/hpx/issues/574
4302 https://github.com/STEllAR-GROUP/hpx/issues/571
4303 https://github.com/STEllAR-GROUP/hpx/issues/570
4304 https://github.com/STEllAR-GROUP/hpx/issues/569
4305 https://github.com/STEllAR-GROUP/hpx/issues/568
4306 https://github.com/STEllAR-GROUP/hpx/issues/567
4307 https://github.com/STEllAR-GROUP/hpx/issues/565
4308 https://github.com/STEllAR-GROUP/hpx/issues/564
4309 https://github.com/STEllAR-GROUP/hpx/issues/563
4310 https://github.com/STEllAR-GROUP/hpx/issues/562
4311 https://github.com/STEllAR-GROUP/hpx/issues/561
4312 https://github.com/STEllAR-GROUP/hpx/issues/558
4313 https://github.com/STEllAR-GROUP/hpx/issues/556
4314 https://github.com/STEllAR-GROUP/hpx/issues/555
4315 https://github.com/STEllAR-GROUP/hpx/issues/554
4316 https://github.com/STEllAR-GROUP/hpx/issues/552
4317 https://github.com/STEllAR-GROUP/hpx/issues/550
4318 https://github.com/STEllAR-GROUP/hpx/issues/549
4319 https://github.com/STEllAR-GROUP/hpx/issues/545
4320 https://github.com/STEllAR-GROUP/hpx/issues/542

- Issue #541[4321] - Quotes around application name in hpx::init
- Issue #539[4322] - Race condititon occurring with new lightweight threads
- Issue #535[4323] - hpx_run_tests.py exits with no error code when tests are missing
- Issue #530[4324] - Thread description(<unknown>) in logs
- Issue #523[4325] - Make thread objects more lightweight
- Issue #521[4326] - `hpx::error_code` is not usable for lightweight error handling
- Issue #520[4327] - Add full user environment to HPX logs
- Issue #519[4328] - Build succeeds, running fails
- Issue #517[4329] - Add a guard page to linux coroutine stacks
- Issue #516[4330] - hpx::thread::detach suspends while holding locks, leads to hang in debug
- Issue #514[4331] - Preprocessed headers for <hpx/apply.hpp> don't compile
- Issue #513[4332] - Buildbot configuration problem
- Issue #512[4333] - Implement action based stack size customization
- Issue #511[4334] - Move action priority into a separate type trait
- Issue #510[4335] - trunk broken
- Issue #507[4336] - no matching function for call to `boost::scoped_ptr<hpx::threads::topology>::scoped_ptr(h`
- Issue #505[4337] - undefined_symbol regression test currently failing
- Issue #502[4338] - Adding OpenCL and OCLM support to HPX for Windows and Linux
- Issue #501[4339] - find_package(HPX) sets cmake output variables
- Issue #500[4340] - wait_any/wait_all are badly named
- Issue #499[4341] - Add support for disabling pbs support in pbs runs
- Issue #498[4342] - Error during no-cache runs
- Issue #496[4343] - Add partial preprocessing support to cmake

[4321] https://github.com/STEllAR-GROUP/hpx/issues/541
[4322] https://github.com/STEllAR-GROUP/hpx/issues/539
[4323] https://github.com/STEllAR-GROUP/hpx/issues/535
[4324] https://github.com/STEllAR-GROUP/hpx/issues/530
[4325] https://github.com/STEllAR-GROUP/hpx/issues/523
[4326] https://github.com/STEllAR-GROUP/hpx/issues/521
[4327] https://github.com/STEllAR-GROUP/hpx/issues/520
[4328] https://github.com/STEllAR-GROUP/hpx/issues/519
[4329] https://github.com/STEllAR-GROUP/hpx/issues/517
[4330] https://github.com/STEllAR-GROUP/hpx/issues/516
[4331] https://github.com/STEllAR-GROUP/hpx/issues/514
[4332] https://github.com/STEllAR-GROUP/hpx/issues/513
[4333] https://github.com/STEllAR-GROUP/hpx/issues/512
[4334] https://github.com/STEllAR-GROUP/hpx/issues/511
[4335] https://github.com/STEllAR-GROUP/hpx/issues/510
[4336] https://github.com/STEllAR-GROUP/hpx/issues/507
[4337] https://github.com/STEllAR-GROUP/hpx/issues/505
[4338] https://github.com/STEllAR-GROUP/hpx/issues/502
[4339] https://github.com/STEllAR-GROUP/hpx/issues/501
[4340] https://github.com/STEllAR-GROUP/hpx/issues/500
[4341] https://github.com/STEllAR-GROUP/hpx/issues/499
[4342] https://github.com/STEllAR-GROUP/hpx/issues/498
[4343] https://github.com/STEllAR-GROUP/hpx/issues/496

- Issue #495[4344] - Support HPX modules exporting startup/shutdown functions only
- Issue #494[4345] - Allow modules to specify when to run startup/shutdown functions
- Issue #493[4346] - Avoid constructing a string in make_success_code
- Issue #492[4347] - Performance counter creation is no longer synchronized at startup
- Issue #491[4348] - Performance counter creation is no longer synchronized at startup
- Issue #490[4349] - Sheneos on_completed_bulk seg fault in distributed
- Issue #489[4350] - compiling issue with g++44
- Issue #488[4351] - Adding OpenCL and OCLM support to HPX for the MSVC platform
- Issue #487[4352] - FindHPX.cmake problems
- Issue #485[4353] - Change distributing_factory and binpacking_factory to use bulk creation
- Issue #484[4354] - Change `HPX_DONT_USE_PREPROCESSED_FILES` to `HPX_USE_PREPROCESSED_FILES`
- Issue #483[4355] - Memory counter for Windows
- Issue #479[4356] - strange errors appear when requesting performance counters on multiple nodes
- Issue #477[4357] - Create (global) timer for multi-threaded measurements
- Issue #472[4358] - Add partial preprocessing using Wave
- Issue #471[4359] - Segfault stack traces don't show up in release
- Issue #468[4360] - External projects need to link with internal components
- Issue #462[4361] - Startup/shutdown functions are called more than once
- Issue #458[4362] - Consolidate hpx::util::high_resolution_timer and `hpx::util::high_resolution_clock`
- Issue #457[4363] - index out of bounds in `allgather_and_gate` on 4 cores or more
- Issue #448[4364] - Make HPX compile with clang
- Issue #447[4365] - 'make tests' should execute tests on local installation
- Issue #446[4366] - Remove SVN-related code from the codebase

4344 https://github.com/STEllAR-GROUP/hpx/issues/495
4345 https://github.com/STEllAR-GROUP/hpx/issues/494
4346 https://github.com/STEllAR-GROUP/hpx/issues/493
4347 https://github.com/STEllAR-GROUP/hpx/issues/492
4348 https://github.com/STEllAR-GROUP/hpx/issues/491
4349 https://github.com/STEllAR-GROUP/hpx/issues/490
4350 https://github.com/STEllAR-GROUP/hpx/issues/489
4351 https://github.com/STEllAR-GROUP/hpx/issues/488
4352 https://github.com/STEllAR-GROUP/hpx/issues/487
4353 https://github.com/STEllAR-GROUP/hpx/issues/485
4354 https://github.com/STEllAR-GROUP/hpx/issues/484
4355 https://github.com/STEllAR-GROUP/hpx/issues/483
4356 https://github.com/STEllAR-GROUP/hpx/issues/479
4357 https://github.com/STEllAR-GROUP/hpx/issues/477
4358 https://github.com/STEllAR-GROUP/hpx/issues/472
4359 https://github.com/STEllAR-GROUP/hpx/issues/471
4360 https://github.com/STEllAR-GROUP/hpx/issues/468
4361 https://github.com/STEllAR-GROUP/hpx/issues/462
4362 https://github.com/STEllAR-GROUP/hpx/issues/458
4363 https://github.com/STEllAR-GROUP/hpx/issues/457
4364 https://github.com/STEllAR-GROUP/hpx/issues/448
4365 https://github.com/STEllAR-GROUP/hpx/issues/447
4366 https://github.com/STEllAR-GROUP/hpx/issues/446

- Issue #444[4367] - race condition in smp
- Issue #441[4368] - Patched Boost.Serialization headers should only be installed if needed
- Issue #439[4369] - Components using `HPX_REGISTER_STARTUP_MODULE` fail to compile with MSVC
- Issue #436[4370] - Verify that no locks are being held while threads are suspended
- Issue #435[4371] - Installing HPX should not clobber existing Boost installation
- Issue #434[4372] - Logging external component failed (Boost 1.50)
- Issue #433[4373] - Runtime crash when building all examples
- Issue #432[4374] - Dataflow hangs on 512 cores/64 nodes
- Issue #430[4375] - Problem with distributing factory
- Issue #424[4376] - File paths referring to XSL-files need to be properly escaped
- Issue #417[4377] - Make dataflow LCOs work out of the box by using partial preprocessing
- Issue #413[4378] - hpx_svnversion.py fails on Windows
- Issue #412[4379] - Make hpx::error_code equivalent to hpx::exception
- Issue #398[4380] - HPX clobbers out-of-tree application specific CMake variables (specifically `CMAKE_BUILD_TYPE`)
- Issue #394[4381] - Remove code generating random port numbers for network
- Issue #378[4382] - ShenEOS scaling issues
- Issue #354[4383] - Create a coroutines wrapper for Boost.Context
- Issue #349[4384] - Commandline option `--localities=N/-lN` should be necessary only on AGAS locality
- Issue #334[4385] - Add auto_index support to cmake based documentation toolchain
- Issue #318[4386] - Network benchmarks
- Issue #317[4387] - Implement network performance counters
- Issue #310[4388] - Duplicate logging entries
- Issue #230[4389] - Add compile time option to disable thread debugging info

4367 https://github.com/STEllAR-GROUP/hpx/issues/444
4368 https://github.com/STEllAR-GROUP/hpx/issues/441
4369 https://github.com/STEllAR-GROUP/hpx/issues/439
4370 https://github.com/STEllAR-GROUP/hpx/issues/436
4371 https://github.com/STEllAR-GROUP/hpx/issues/435
4372 https://github.com/STEllAR-GROUP/hpx/issues/434
4373 https://github.com/STEllAR-GROUP/hpx/issues/433
4374 https://github.com/STEllAR-GROUP/hpx/issues/432
4375 https://github.com/STEllAR-GROUP/hpx/issues/430
4376 https://github.com/STEllAR-GROUP/hpx/issues/424
4377 https://github.com/STEllAR-GROUP/hpx/issues/417
4378 https://github.com/STEllAR-GROUP/hpx/issues/413
4379 https://github.com/STEllAR-GROUP/hpx/issues/412
4380 https://github.com/STEllAR-GROUP/hpx/issues/398
4381 https://github.com/STEllAR-GROUP/hpx/issues/394
4382 https://github.com/STEllAR-GROUP/hpx/issues/378
4383 https://github.com/STEllAR-GROUP/hpx/issues/354
4384 https://github.com/STEllAR-GROUP/hpx/issues/349
4385 https://github.com/STEllAR-GROUP/hpx/issues/334
4386 https://github.com/STEllAR-GROUP/hpx/issues/318
4387 https://github.com/STEllAR-GROUP/hpx/issues/317
4388 https://github.com/STEllAR-GROUP/hpx/issues/310
4389 https://github.com/STEllAR-GROUP/hpx/issues/230

- Issue #171[4390] - Add an INI option to turn off deadlock detection independently of logging
- Issue #170[4391] - OSHL internal counters are incorrect
- Issue #103[4392] - Better diagnostics for multiple component/action registerations under the same name
- Issue #48[4393] - Support for Darwin (Xcode + Clang)
- Issue #21[4394] - Build fails with GCC 4.6

2.11.18 *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[4395].
- *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.

[4390] https://github.com/STEllAR-GROUP/hpx/issues/171
[4391] https://github.com/STEllAR-GROUP/hpx/issues/170
[4392] https://github.com/STEllAR-GROUP/hpx/issues/103
[4393] https://github.com/STEllAR-GROUP/hpx/issues/48
[4394] https://github.com/STEllAR-GROUP/hpx/issues/21
[4395] http://www.open-std.org/jtc1/sc22/wg21

- 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:

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

int hpx_main()
{
    return hpx::finalize();
}

int main()
{
    return hpx::init();
}
```

This removes the immediate dependency on the Boost.Program Options[4396] 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[4397], the minimal program can be written as:

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

// The arguments for hpx_main can be left off, which very similar to the
// behavior of ``main()`` as defined by C++.
int hpx_main(int argc, char* argv[])
{
    return hpx::finalize();
}

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

- Added performance counters exposing the number of component instances which are alive on a given locality.
- Added performance counters exposing then number of messages sent and received, the number of parcels sent and received, the number of bytes sent and received, the overall time required to send and receive data, and the overall time required to serialize and deserialize the data.
- Added a new component: `hpx::components::binpacking_factory` which is equivalent to the existing `hpx::components::distributing_factory` component, except that it equalizes the overall population of the components to create. It exposes two factory methods, one based on the number of existing instances of the component type to create, and one based on an arbitrary performance counter which will be queried for all relevant localities.

[4396] https://www.boost.org/doc/html/program_options.html
[4397] https://www.boost.org/doc/html/program_options.html

- 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[4398] - GIDs that are not serialized via handle_gid<> should raise an exception
- Issue #105[4399] - Allow for hpx::util::functions to be registered in the AGAS symbolic namespace
- Issue #107[4400] - Nasty threadmanger race condition (reproducible in sheneos_test)
- Issue #108[4401] - Add millisecond resolution to *HPX* logs on Linux
- Issue #110[4402] - Shutdown hang in distributed with release build
- Issue #116[4403] - Don't use TSS for the applier and runtime pointers
- Issue #162[4404] - Move local synchronous execution shortcut from hpx::function to the applier
- Issue #172[4405] - Cache sources in CMake and check if they change manually
- Issue #178[4406] - Add an INI option to turn off ranged-based AGAS caching
- Issue #187[4407] - Support for disabling performance counter deployment
- Issue #202[4408] - Support for sending performance counter data to a specific file
- Issue #218[4409] - boost.coroutines allows different stack sizes, but stack pool is unaware of this
- Issue #231[4410] - Implement movable boost::bind
- Issue #232[4411] - Implement movable boost::function
- Issue #236[4412] - Allow binding hpx::util::function to actions
- Issue #239[4413] - Replace hpx::function with hpx::util::function
- Issue #240[4414] - Can't specify RemoteResult with lcos::async
- Issue #242[4415] - REGISTER_TEMPLATE support for plain actions
- Issue #243[4416] - handle_gid<> support for hpx::util::function

[4398] https://github.com/STEllAR-GROUP/hpx/issues/71
[4399] https://github.com/STEllAR-GROUP/hpx/issues/105
[4400] https://github.com/STEllAR-GROUP/hpx/issues/107
[4401] https://github.com/STEllAR-GROUP/hpx/issues/108
[4402] https://github.com/STEllAR-GROUP/hpx/issues/110
[4403] https://github.com/STEllAR-GROUP/hpx/issues/116
[4404] https://github.com/STEllAR-GROUP/hpx/issues/162
[4405] https://github.com/STEllAR-GROUP/hpx/issues/172
[4406] https://github.com/STEllAR-GROUP/hpx/issues/178
[4407] https://github.com/STEllAR-GROUP/hpx/issues/187
[4408] https://github.com/STEllAR-GROUP/hpx/issues/202
[4409] https://github.com/STEllAR-GROUP/hpx/issues/218
[4410] https://github.com/STEllAR-GROUP/hpx/issues/231
[4411] https://github.com/STEllAR-GROUP/hpx/issues/232
[4412] https://github.com/STEllAR-GROUP/hpx/issues/236
[4413] https://github.com/STEllAR-GROUP/hpx/issues/239
[4414] https://github.com/STEllAR-GROUP/hpx/issues/240
[4415] https://github.com/STEllAR-GROUP/hpx/issues/242
[4416] https://github.com/STEllAR-GROUP/hpx/issues/243

- Issue #245[4417] - `*_c_cache code` throws an exception if the queried GID is not in the local cache
- Issue #246[4418] - Undefined references in dataflow/adaptive1d example
- Issue #252[4419] - Problems configuring sheneos with CMake
- Issue #254[4420] - Lifetime of components doesn't end when client goes out of scope
- Issue #259[4421] - CMake does not detect that MSVC10 has lambdas
- Issue #260[4422] - io_service_pool segfault
- Issue #261[4423] - Late parcel executed outside of pxthread
- Issue #263[4424] - Cannot select allocator with CMake
- Issue #264[4425] - Fix allocator select
- Issue #267[4426] - Runtime error for hello_world
- Issue #269[4427] - pthread_affinity_np test fails to compile
- Issue #270[4428] - Compiler noise due to -Wcast-qual
- Issue #275[4429] - Problem with configuration tests/include paths on Gentoo
- Issue #325[4430] - Sheneos is 200-400 times slower than the fortran equivalent
- Issue #331[4431] - *`hpx::init`* and `hpx_main()` should not depend on program_options
- Issue #333[4432] - Add doxygen support to CMake for doc toolchain
- Issue #340[4433] - Performance counters for parcels
- Issue #346[4434] - Component loading error when running hello_world in distributed on MSVC2010
- Issue #362[4435] - Missing initializer error
- Issue #363[4436] - Parcel port serialization error
- Issue #366[4437] - Parcel buffering leads to types incompatible exception
- Issue #368[4438] - Scalable alternative to rand() needed for *HPX*
- Issue #369[4439] - IB over IP is substantially slower than just using standard TCP/IP

4417 https://github.com/STEllAR-GROUP/hpx/issues/245
4418 https://github.com/STEllAR-GROUP/hpx/issues/246
4419 https://github.com/STEllAR-GROUP/hpx/issues/252
4420 https://github.com/STEllAR-GROUP/hpx/issues/254
4421 https://github.com/STEllAR-GROUP/hpx/issues/259
4422 https://github.com/STEllAR-GROUP/hpx/issues/260
4423 https://github.com/STEllAR-GROUP/hpx/issues/261
4424 https://github.com/STEllAR-GROUP/hpx/issues/263
4425 https://github.com/STEllAR-GROUP/hpx/issues/264
4426 https://github.com/STEllAR-GROUP/hpx/issues/267
4427 https://github.com/STEllAR-GROUP/hpx/issues/269
4428 https://github.com/STEllAR-GROUP/hpx/issues/270
4429 https://github.com/STEllAR-GROUP/hpx/issues/275
4430 https://github.com/STEllAR-GROUP/hpx/issues/325
4431 https://github.com/STEllAR-GROUP/hpx/issues/331
4432 https://github.com/STEllAR-GROUP/hpx/issues/333
4433 https://github.com/STEllAR-GROUP/hpx/issues/340
4434 https://github.com/STEllAR-GROUP/hpx/issues/346
4435 https://github.com/STEllAR-GROUP/hpx/issues/362
4436 https://github.com/STEllAR-GROUP/hpx/issues/363
4437 https://github.com/STEllAR-GROUP/hpx/issues/366
4438 https://github.com/STEllAR-GROUP/hpx/issues/368
4439 https://github.com/STEllAR-GROUP/hpx/issues/369

- Issue #374[4440] - `hpx::lcos::wait` should work with dataflows and arbitrary classes meeting the future interface
- Issue #375[4441] - Conflicting/ambiguous overloads of `hpx::lcos::wait`
- Issue #376[4442] - Find_HPX.cmake should set CMake variable HPX_FOUND for out of tree builds
- Issue #377[4443] - ShenEOS interpolate bulk and interpolate_one_bulk are broken
- Issue #379[4444] - Add support for distributed runs under SLURM
- Issue #382[4445] - _Unwind_Word not declared in boost.backtrace
- Issue #387[4446] - Doxygen should look only at list of specified files
- Issue #388[4447] - Running `make install` on an out-of-tree application is broken
- Issue #391[4448] - Out-of-tree application segfaults when running in qsub
- Issue #392[4449] - Remove HPX_NO_INSTALL option from cmake build system
- Issue #396[4450] - Pragma related warnings when compiling with older gcc versions
- Issue #399[4451] - Out of tree component build problems
- Issue #400[4452] - Out of source builds on Windows: linker should not receive compiler flags
- Issue #401[4453] - Out of source builds on Windows: components need to be linked with hpx_serialization
- Issue #404[4454] - gfortran fails to link automatically when fortran files are present
- Issue #405[4455] - Inability to specify linking order for external libraries
- Issue #406[4456] - Adapt action limits such that dataflow applications work without additional defines
- Issue #415[4457] - `locality_results` is not a member of `hpx::components::server`
- Issue #425[4458] - Breaking changes to `traits::*result` wrt `std::vector<id_type>`
- Issue #426[4459] - AUTOGLOB needs to be updated to support fortran

4440 https://github.com/STEllAR-GROUP/hpx/issues/374
4441 https://github.com/STEllAR-GROUP/hpx/issues/375
4442 https://github.com/STEllAR-GROUP/hpx/issues/376
4443 https://github.com/STEllAR-GROUP/hpx/issues/377
4444 https://github.com/STEllAR-GROUP/hpx/issues/379
4445 https://github.com/STEllAR-GROUP/hpx/issues/382
4446 https://github.com/STEllAR-GROUP/hpx/issues/387
4447 https://github.com/STEllAR-GROUP/hpx/issues/388
4448 https://github.com/STEllAR-GROUP/hpx/issues/391
4449 https://github.com/STEllAR-GROUP/hpx/issues/392
4450 https://github.com/STEllAR-GROUP/hpx/issues/396
4451 https://github.com/STEllAR-GROUP/hpx/issues/399
4452 https://github.com/STEllAR-GROUP/hpx/issues/400
4453 https://github.com/STEllAR-GROUP/hpx/issues/401
4454 https://github.com/STEllAR-GROUP/hpx/issues/404
4455 https://github.com/STEllAR-GROUP/hpx/issues/405
4456 https://github.com/STEllAR-GROUP/hpx/issues/406
4457 https://github.com/STEllAR-GROUP/hpx/issues/415
4458 https://github.com/STEllAR-GROUP/hpx/issues/425
4459 https://github.com/STEllAR-GROUP/hpx/issues/426

2.11.19 *HPX* V0.8.1 (Apr 21, 2012)

This is a point release including important bug fixes for *HPX V0.8.0 (Mar 23, 2012)*.

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[4460] - Don't require install path to be known at compile time.
- Issue #371[4461] - Add hpx iostreams to standard build.
- Issue #384[4462] - Fix compilation with GCC 4.7.
- Issue #390[4463] - Remove keep_factory_alive startup call from ShenEOS; add shutdown call to H5close.
- Issue #393[4464] - 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.

2.11.20 *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`).

4460 https://github.com/STEllAR-GROUP/hpx/issues/295
4461 https://github.com/STEllAR-GROUP/hpx/issues/371
4462 https://github.com/STEllAR-GROUP/hpx/issues/384
4463 https://github.com/STEllAR-GROUP/hpx/issues/390
4464 https://github.com/STEllAR-GROUP/hpx/issues/393

- 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.
- 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[4465] (thanks to Matthew Anderson for contributing this application).
- GTC (Gyrokinetic Toroidal Code)[4466]: a skeleton for particle in cell type codes.
- Random Memory Access: an example demonstrating random memory accesses in a large array
- ShenEOS example[4467], 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.

4465 http://www.graph500.org/
4466 http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/gtc/
4467 http://stellarcollapse.org/equationofstate

- Quickstart examples (like hello_world, fibonacci, quicksort, factorial, etc.) demonstrating simple *HPX* concepts which introduce some of the concepts in *HPX*.
- Load balancing and work stealing demos.

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[4468] - Specialize handle_gid<> for examples and tests
- Issue #72[4469] - Fix AGAS reference counting
- Issue #104[4470] - heartbeat throws an exception when decrefing the performance counter it's watching
- Issue #111[4471] - throttle causes an exception on the target application
- Issue #142[4472] - One failed component loading causes an unrelated component to fail
- Issue #165[4473] - Remote exception propagation bug in AGAS reference counting test
- Issue #186[4474] - Test credit exhaustion/splitting (e.g. prepare_gid and symbol NS)
- Issue #188[4475] - Implement remaining AGAS reference counting test cases

4468 https://github.com/STEllAR-GROUP/hpx/issues/31
4469 https://github.com/STEllAR-GROUP/hpx/issues/72
4470 https://github.com/STEllAR-GROUP/hpx/issues/104
4471 https://github.com/STEllAR-GROUP/hpx/issues/111
4472 https://github.com/STEllAR-GROUP/hpx/issues/142
4473 https://github.com/STEllAR-GROUP/hpx/issues/165
4474 https://github.com/STEllAR-GROUP/hpx/issues/186
4475 https://github.com/STEllAR-GROUP/hpx/issues/188

- Issue #258[4476] - No type checking of GIDs in stubs classes
- Issue #271[4477] - Seg fault/shared pointer assertion in distributed code
- Issue #281[4478] - CMake options need descriptive text
- Issue #283[4479] - AGAS caching broken (gva_cache needs to be rewritten with ICL)
- Issue #285[4480] - HPX_INSTALL root directory not the same as CMAKE_INSTALL_PREFIX
- Issue #286[4481] - New segfault in dataflow applications
- Issue #289[4482] - Exceptions should only be logged if not handled
- Issue #290[4483] - c++11 tests failure
- Issue #293[4484] - Build target for component libraries
- Issue #296[4485] - Compilation error with Boost V1.49rc1
- Issue #298[4486] - Illegal instructions on termination
- Issue #299[4487] - gravity aborts with multiple threads
- Issue #301[4488] - Build error with Boost trunk
- Issue #303[4489] - Logging assertion failure in distributed runs
- Issue #304[4490] - Exception 'what' strings are lost when exceptions from decode_parcel are reported
- Issue #306[4491] - Performance counter user interface issues
- Issue #307[4492] - Logging exception in distributed runs
- Issue #308[4493] - Logging deadlocks in distributed
- Issue #309[4494] - Reference counting test failures and exceptions
- Issue #311[4495] - Merge AGAS remote_interface with the runtime_support object
- Issue #314[4496] - Object tracking for id_types
- Issue #315[4497] - Remove handle_gid and handle credit splitting in id_type serialization
- Issue #320[4498] - applier::get_locality_id() should return an error value (or throw an exception)

4476 https://github.com/STEllAR-GROUP/hpx/issues/258
4477 https://github.com/STEllAR-GROUP/hpx/issues/271
4478 https://github.com/STEllAR-GROUP/hpx/issues/281
4479 https://github.com/STEllAR-GROUP/hpx/issues/283
4480 https://github.com/STEllAR-GROUP/hpx/issues/285
4481 https://github.com/STEllAR-GROUP/hpx/issues/286
4482 https://github.com/STEllAR-GROUP/hpx/issues/289
4483 https://github.com/STEllAR-GROUP/hpx/issues/290
4484 https://github.com/STEllAR-GROUP/hpx/issues/293
4485 https://github.com/STEllAR-GROUP/hpx/issues/296
4486 https://github.com/STEllAR-GROUP/hpx/issues/298
4487 https://github.com/STEllAR-GROUP/hpx/issues/299
4488 https://github.com/STEllAR-GROUP/hpx/issues/301
4489 https://github.com/STEllAR-GROUP/hpx/issues/303
4490 https://github.com/STEllAR-GROUP/hpx/issues/304
4491 https://github.com/STEllAR-GROUP/hpx/issues/306
4492 https://github.com/STEllAR-GROUP/hpx/issues/307
4493 https://github.com/STEllAR-GROUP/hpx/issues/308
4494 https://github.com/STEllAR-GROUP/hpx/issues/309
4495 https://github.com/STEllAR-GROUP/hpx/issues/311
4496 https://github.com/STEllAR-GROUP/hpx/issues/314
4497 https://github.com/STEllAR-GROUP/hpx/issues/315
4498 https://github.com/STEllAR-GROUP/hpx/issues/320

- Issue #321[4499] - Optimization for id_types which are never split should be restored
- Issue #322[4500] - Command line processing ignored with Boost 1.47.0
- Issue #323[4501] - Credit exhaustion causes object to stay alive
- Issue #324[4502] - Duplicate exception messages
- Issue #326[4503] - Integrate Quickbook with CMake
- Issue #329[4504] - –help and –version should still work
- Issue #330[4505] - Create pkg-config files
- Issue #337[4506] - Improve usability of performance counter timestamps
- Issue #338[4507] - Non-std exceptions deriving from std::exceptions in tfunc may be sliced
- Issue #339[4508] - Decrease the number of send_pending_parcels threads
- Issue #343[4509] - Dynamically setting the stack size doesn't work
- Issue #351[4510] - 'make install' does not update documents
- Issue #353[4511] - Disable FIXMEs in the docs by default; add a doc developer CMake option to enable FIXMEs
- Issue #355[4512] - 'make' doesn't do anything after correct configuration
- Issue #356[4513] - Don't use `hpx::util::static_` in topology code
- Issue #359[4514] - Infinite recursion in hpx::tuple serialization
- Issue #361[4515] - Add compile time option to disable logging completely
- Issue #364[4516] - Installation seriously broken in r7443

2.11.21 *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.).

[4499] https://github.com/STEllAR-GROUP/hpx/issues/321
[4500] https://github.com/STEllAR-GROUP/hpx/issues/322
[4501] https://github.com/STEllAR-GROUP/hpx/issues/323
[4502] https://github.com/STEllAR-GROUP/hpx/issues/324
[4503] https://github.com/STEllAR-GROUP/hpx/issues/326
[4504] https://github.com/STEllAR-GROUP/hpx/issues/329
[4505] https://github.com/STEllAR-GROUP/hpx/issues/330
[4506] https://github.com/STEllAR-GROUP/hpx/issues/337
[4507] https://github.com/STEllAR-GROUP/hpx/issues/338
[4508] https://github.com/STEllAR-GROUP/hpx/issues/339
[4509] https://github.com/STEllAR-GROUP/hpx/issues/343
[4510] https://github.com/STEllAR-GROUP/hpx/issues/351
[4511] https://github.com/STEllAR-GROUP/hpx/issues/353
[4512] https://github.com/STEllAR-GROUP/hpx/issues/355
[4513] https://github.com/STEllAR-GROUP/hpx/issues/356
[4514] https://github.com/STEllAR-GROUP/hpx/issues/359
[4515] https://github.com/STEllAR-GROUP/hpx/issues/361
[4516] 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[4517] - Integrate Windows/Linux CMake code for *HPX* core
- Issue #32[4518] - hpx::cout() should be hpx::cout
- Issue #33[4519] - AGAS V2 legacy client does not properly handle error_code
- Issue #60[4520] - AGAS: allow for registerid to optionally take ownership of the gid
- Issue #62[4521] - adaptive1d compilation failure in Fusion
- Issue #64[4522] - Parcel subsystem doesn't resolve domain names
- Issue #83[4523] - No error handling if no console is available
- Issue #84[4524] - No error handling if a hosted locality is treated as the bootstrap server
- Issue #90[4525] - Add general commandline option -N
- Issue #91[4526] - Add possibility to read command line arguments from file
- Issue #92[4527] - Always log exceptions/errors to the log file
- Issue #93[4528] - Log the command line/program name
- Issue #95[4529] - Support for distributed launches
- Issue #97[4530] - Attempt to create a bad component type in AMR examples
- Issue #100[4531] - factorial and factorial_get examples trigger AGAS component type assertions
- Issue #101[4532] - Segfault when hpx::process::here() is called in fibonacci2
- Issue #102[4533] - unknown_component_address in int_object_semaphore_client
- Issue #114[4534] - marduk raises assertion with default parameters
- Issue #115[4535] - Logging messages for SMP runs (on the console) shouldn't be buffered

[4517] https://github.com/STEllAR-GROUP/hpx/issues/28
[4518] https://github.com/STEllAR-GROUP/hpx/issues/32
[4519] https://github.com/STEllAR-GROUP/hpx/issues/33
[4520] https://github.com/STEllAR-GROUP/hpx/issues/60
[4521] https://github.com/STEllAR-GROUP/hpx/issues/62
[4522] https://github.com/STEllAR-GROUP/hpx/issues/64
[4523] https://github.com/STEllAR-GROUP/hpx/issues/83
[4524] https://github.com/STEllAR-GROUP/hpx/issues/84
[4525] https://github.com/STEllAR-GROUP/hpx/issues/90
[4526] https://github.com/STEllAR-GROUP/hpx/issues/91
[4527] https://github.com/STEllAR-GROUP/hpx/issues/92
[4528] https://github.com/STEllAR-GROUP/hpx/issues/93
[4529] https://github.com/STEllAR-GROUP/hpx/issues/95
[4530] https://github.com/STEllAR-GROUP/hpx/issues/97
[4531] https://github.com/STEllAR-GROUP/hpx/issues/100
[4532] https://github.com/STEllAR-GROUP/hpx/issues/101
[4533] https://github.com/STEllAR-GROUP/hpx/issues/102
[4534] https://github.com/STEllAR-GROUP/hpx/issues/114
[4535] https://github.com/STEllAR-GROUP/hpx/issues/115

- Issue #119[4536] - marduk linking strategy breaks other applications
- Issue #121[4537] - pbsdsh problem
- Issue #123[4538] - marduk, dataflow and adaptive1d fail to build
- Issue #124[4539] - Lower default preprocessing arity
- Issue #125[4540] - Move hpx::detail::diagnostic_information out of the detail namespace
- Issue #126[4541] - Test definitions for AGAS reference counting
- Issue #128[4542] - Add averaging performance counter
- Issue #129[4543] - Error with endian.hpp while building adaptive1d
- Issue #130[4544] - Bad initialization of performance counters
- Issue #131[4545] - Add global startup/shutdown functions to component modules
- Issue #132[4546] - Avoid using auto_ptr
- Issue #133[4547] - On Windows hpx.dll doesn't get installed
- Issue #134[4548] - HPX_LIBRARY does not reflect real library name (on Windows)
- Issue #135[4549] - Add detection of unique_ptr to build system
- Issue #137[4550] - Add command line option allowing to repeatedly evaluate performance counters
- Issue #139[4551] - Logging is broken
- Issue #140[4552] - CMake problem on windows
- Issue #141[4553] - Move all non-component libraries into $PREFIX/lib/hpx
- Issue #143[4554] - adaptive1d throws an exception with the default command line options
- Issue #146[4555] - Early exception handling is broken
- Issue #147[4556] - Sheneos doesn't link on Linux
- Issue #149[4557] - sheneos_test hangs
- Issue #154[4558] - Compilation fails for r5661

4536 https://github.com/STEllAR-GROUP/hpx/issues/119
4537 https://github.com/STEllAR-GROUP/hpx/issues/121
4538 https://github.com/STEllAR-GROUP/hpx/issues/123
4539 https://github.com/STEllAR-GROUP/hpx/issues/124
4540 https://github.com/STEllAR-GROUP/hpx/issues/125
4541 https://github.com/STEllAR-GROUP/hpx/issues/126
4542 https://github.com/STEllAR-GROUP/hpx/issues/128
4543 https://github.com/STEllAR-GROUP/hpx/issues/129
4544 https://github.com/STEllAR-GROUP/hpx/issues/130
4545 https://github.com/STEllAR-GROUP/hpx/issues/131
4546 https://github.com/STEllAR-GROUP/hpx/issues/132
4547 https://github.com/STEllAR-GROUP/hpx/issues/133
4548 https://github.com/STEllAR-GROUP/hpx/issues/134
4549 https://github.com/STEllAR-GROUP/hpx/issues/135
4550 https://github.com/STEllAR-GROUP/hpx/issues/137
4551 https://github.com/STEllAR-GROUP/hpx/issues/139
4552 https://github.com/STEllAR-GROUP/hpx/issues/140
4553 https://github.com/STEllAR-GROUP/hpx/issues/141
4554 https://github.com/STEllAR-GROUP/hpx/issues/143
4555 https://github.com/STEllAR-GROUP/hpx/issues/146
4556 https://github.com/STEllAR-GROUP/hpx/issues/147
4557 https://github.com/STEllAR-GROUP/hpx/issues/149
4558 https://github.com/STEllAR-GROUP/hpx/issues/154

- Issue #155[4559] - Sine performance counters example chokes on chrono headers
- Issue #156[4560] - Add build type to –version
- Issue #157[4561] - Extend AGAS caching to store gid ranges
- Issue #158[4562] - r5691 doesn't compile
- Issue #160[4563] - Re-add AGAS function for resolving a locality to its prefix
- Issue #168[4564] - Managed components should be able to access their own GID
- Issue #169[4565] - Rewrite AGAS future pool
- Issue #179[4566] - Complete switch to request class for AGAS server interface
- Issue #182[4567] - Sine performance counter is loaded by other examples
- Issue #185[4568] - Write tests for symbol namespace reference counting
- Issue #191[4569] - Assignment of read-only variable in point_geometry
- Issue #200[4570] - Seg faults when querying performance counters
- Issue #204[4571] - –ifnames and suffix stripping needs to be more generic
- Issue #205[4572] - –list-* and –print-counter-* options do not work together and produce no warning
- Issue #207[4573] - Implement decrement entry merging
- Issue #208[4574] - Replace the spinlocks in AGAS with hpx::lcos::local_mutexes
- Issue #210[4575] - Add an –ifprefix option
- Issue #214[4576] - Performance test for PX-thread creation
- Issue #216[4577] - VS2010 compilation
- Issue #222[4578] - r6045 context_linux_x86.hpp
- Issue #223[4579] - fibonacci hangs when changing the state of an active thread
- Issue #225[4580] - Active threads end up in the FEB wait queue
- Issue #226[4581] - VS Build Error for Accumulator Client

[4559] https://github.com/STEllAR-GROUP/hpx/issues/155
[4560] https://github.com/STEllAR-GROUP/hpx/issues/156
[4561] https://github.com/STEllAR-GROUP/hpx/issues/157
[4562] https://github.com/STEllAR-GROUP/hpx/issues/158
[4563] https://github.com/STEllAR-GROUP/hpx/issues/160
[4564] https://github.com/STEllAR-GROUP/hpx/issues/168
[4565] https://github.com/STEllAR-GROUP/hpx/issues/169
[4566] https://github.com/STEllAR-GROUP/hpx/issues/179
[4567] https://github.com/STEllAR-GROUP/hpx/issues/182
[4568] https://github.com/STEllAR-GROUP/hpx/issues/185
[4569] https://github.com/STEllAR-GROUP/hpx/issues/191
[4570] https://github.com/STEllAR-GROUP/hpx/issues/200
[4571] https://github.com/STEllAR-GROUP/hpx/issues/204
[4572] https://github.com/STEllAR-GROUP/hpx/issues/205
[4573] https://github.com/STEllAR-GROUP/hpx/issues/207
[4574] https://github.com/STEllAR-GROUP/hpx/issues/208
[4575] https://github.com/STEllAR-GROUP/hpx/issues/210
[4576] https://github.com/STEllAR-GROUP/hpx/issues/214
[4577] https://github.com/STEllAR-GROUP/hpx/issues/216
[4578] https://github.com/STEllAR-GROUP/hpx/issues/222
[4579] https://github.com/STEllAR-GROUP/hpx/issues/223
[4580] https://github.com/STEllAR-GROUP/hpx/issues/225
[4581] https://github.com/STEllAR-GROUP/hpx/issues/226

- Issue #228[4582] - Move all traits into namespace hpx::traits
- Issue #229[4583] - Invalid initialization of reference in thread_init_data
- Issue #235[4584] - Invalid GID in iostreams
- Issue #238[4585] - Demangle type names for the default implementation of get_action_name
- Issue #241[4586] - C++11 support breaks GCC 4.5
- Issue #247[4587] - Reference to temporary with GCC 4.4
- Issue #248[4588] - Seg fault at shutdown with GCC 4.4
- Issue #253[4589] - Default component action registration kills compiler
- Issue #272[4590] - G++ unrecognized command line option
- Issue #273[4591] - quicksort example doesn't compile
- Issue #277[4592] - Invalid CMake logic for Windows

2.12 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*: [4593]. Use the Zenodo entry for referring to the latest version of *HPX*: [4594]. Entries for citing specific versions of *HPX* can also be found at [4595].

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)[4596], a multi-disciplinary research institute at Louisiana State University (LSU)[4597]. The ParalleX group was working to develop a new and experimental execution model for future high performance computing architectures. This model was christened ParalleX. The first implementations of ParalleX were crude, and many of those designs had to be discarded entirely. However, over time the team learned quite a bit about how to design a parallel, distributed runtime system which implements the concepts of ParalleX.

4582 https://github.com/STEllAR-GROUP/hpx/issues/228
4583 https://github.com/STEllAR-GROUP/hpx/issues/229
4584 https://github.com/STEllAR-GROUP/hpx/issues/235
4585 https://github.com/STEllAR-GROUP/hpx/issues/238
4586 https://github.com/STEllAR-GROUP/hpx/issues/241
4587 https://github.com/STEllAR-GROUP/hpx/issues/247
4588 https://github.com/STEllAR-GROUP/hpx/issues/248
4589 https://github.com/STEllAR-GROUP/hpx/issues/253
4590 https://github.com/STEllAR-GROUP/hpx/issues/272
4591 https://github.com/STEllAR-GROUP/hpx/issues/273
4592 https://github.com/STEllAR-GROUP/hpx/issues/277
4593 https://joss.theoj.org/papers/022e5917b95517dff20cd3742ab95eca
4594 https://doi.org/10.5281/zenodo.598202
4595 https://doi.org/10.5281/zenodo.598202
4596 https://www.cct.lsu.edu
4597 https://www.lsu.edu

From the very beginning, this endeavour has been a group effort. In addition to a handful of interested researchers, there have always been graduate and undergraduate students participating in the discussions, design, and implementation of *HPX*. In 2011 we decided to formalize our collective research efforts by creating the STE||AR[4598] group (**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch). Over time, the team grew to include researchers around the country and the world. In 2014, the STE||AR[4599] 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++11/14 ISO standard and adheres to the programming guidelines used by the Boost[4600] collection of C++ libraries. We steer the development of *HPX* with real world applications and aim to provide a smooth migration path for domain scientists.

To learn more about STE||AR[4601] and ParalleX, see *People* and *Why HPX?*.

2.13.2 People

The STE||AR[4602] Group (pronounced as stellar) stands for "**S**ystems **T**echnology, **E**mergent Para**ll**elism, and **A**lgorithm **R**esearch". We are an international group of faculty, researchers, and students working at various institutions around the world. The goal of the STE||AR[4603] 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.

[4598] https://stellar-group.org
[4599] https://stellar-group.org
[4600] https://www.boost.org/
[4601] https://stellar-group.org
[4602] https://stellar-group.org
[4603] https://stellar-group.org

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 2.39: Contributors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT)[4604], Louisiana State University (LSU)[4605]	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture[4606], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[4607]	thom.heller@gmail.com
Agustin Berge	Center for Computation and Technology (CCT)[4608], Louisiana State University (LSU)[4609]	kaballo86@hotmail.com
Mikael Simberg	Swiss National Supercomputing Centre[4610]	simbergm@cscs.ch
John Biddiscombe	Swiss National Supercomputing Centre[4611]	biddisco@cscs.ch
Anton Bikineev	Center for Computation and Technology (CCT)[4612], Louisiana State University (LSU)[4613]	ant.bikineev@gmail.com
Martin Stumpf	Department of Computer Science 3 - Computer Architecture[4614], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[4615]	martin.h.stumpf@gmail.com
Bryce Adelstein Lelbach	NVIDIA[4616]	brycelelbach@gmail.com
Shuangyang Yang	Center for Computation and Technology (CCT)[4617], Louisiana State University (LSU)[4618]	syang16@cct.lsu.edu
Jeroen Habraken	Technische Universiteit Eindhoven[4619]	jhabraken@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT)[4620], Louisiana State University (LSU)[4621]	sbrandt@cct.lsu.edu
Antoine Tran Tan	Center for Computation and Technology (CCT)[4622], Louisiana State University (LSU)[4623]	antoine.trantan@lri.fr
Adrian Serio	Center for Computation and Technology (CCT)[4624], Louisiana State University (LSU)[4625]	aserio@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST)[4626], Indiana University (IU)[4627]	mbrodowi@indiana.edu

Contributors to this document

Table 2.40: Documentation authors

Name	Institution	Email
Hartmut Kaiser	Center for Computation and Technology (CCT)[4628], Louisiana State University (LSU)[4629]	hkaiser@cct.lsu.edu
Thomas Heller	Department of Computer Science 3 - Computer Architecture[4630], Friedrich-Alexander University Erlangen-Nuremberg (FAU)[4631]	thom.heller@gmail.com
Bryce Adelstein Lelbach	NVIDIA[4632]	brycelelbach@gmail.com
Vinay C Amatya	Center for Computation and Technology (CCT)[4633], Louisiana State University (LSU)[4634]	vamatya@cct.lsu.edu
Steven Brandt	Center for Computation and Technology (CCT)[4635], Louisiana State University (LSU)[4636]	sbrandt@cct.lsu.edu
Maciej Brodowicz	Center for Research in Extreme Scale Technologies (CREST)[4637], Indiana University (IU)[4638]	mbrodowi@indiana.edu
Adrian Serio	Center for Computation and Technology (CCT)[4639], Louisiana State University (LSU)[4640]	aserio@cct.lsu.edu

4604 https://www.cct.lsu.edu
4605 https://www.lsu.edu
4606 https://www3.cs.fau.de
4607 https://www.fau.de
4608 https://www.cct.lsu.edu
4609 https://www.lsu.edu
4610 https://www.cscs.ch
4611 https://www.cscs.ch
4612 https://www.cct.lsu.edu
4613 https://www.lsu.edu
4614 https://www3.cs.fau.de
4615 https://www.fau.de
4616 https://nvidia.com/
4617 https://www.cct.lsu.edu
4618 https://www.lsu.edu
4619 https://www.tui.nl
4620 https://www.cct.lsu.edu
4621 https://www.lsu.edu
4622 https://www.cct.lsu.edu
4623 https://www.lsu.edu
4624 https://www.cct.lsu.edu
4625 https://www.lsu.edu
4626 https://pti.iu.edu
4627 https://www.iu.edu
4628 https://www.cct.lsu.edu
4629 https://www.lsu.edu
4630 https://www3.cs.fau.de
4631 https://www.fau.de
4632 https://nvidia.com/
4633 https://www.cct.lsu.edu
4634 https://www.lsu.edu
4635 https://www.cct.lsu.edu
4636 https://www.lsu.edu
4637 https://pti.iu.edu
4638 https://www.iu.edu
4639 https://www.cct.lsu.edu
4640 https://www.lsu.edu

Acknowledgements

Thanks also to the following people who contributed directly or indirectly to the project through discussions, pull requests, documentation patches, etc.

- Giannis Gonidelis, for his work on the ranges adaptation during the Google Summer of Code 2020.
- Auriane Reverdell (Swiss National Supercomputing Centre[4641]), 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[4642]), 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.
- 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.

[4641] https://www.cscs.ch
[4642] https://www.cscs.ch

- Zach Byerly (Louisiana State University (LSU)[4643]), 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)[4644]), 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)[4645]), who contributed the prefetching iterators and the persistent auto chunking executor parameters implementation.
- Marcin Copik, who worked on implementing GPU support for C++AMP and HCC. He also worked on implementing a HCC backend for *HPX.Compute*.
- Minh-Khanh Do, who contributed the implementation of several segmented algorithms.
- Bibek Wagle (Louisiana State University (LSU)[4646]), 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[4647]) with *HPX*. He reported various problems and submitted several patches to fix issues allowing for a better integration with LibGeoDecomp[4648].
- 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)[4649]), 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[4650] C/C++ package manager.
- Vinay C Amatya (Louisiana State University (LSU)[4651]), who contributed to the documentation and provided some of the *HPX* examples.
- Kevin Huck and Nick Chaimov (University of Oregon[4652]), 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*.

[4643] https://www.lsu.edu
[4644] https://www.lsu.edu
[4645] https://www.lsu.edu
[4646] https://www.lsu.edu
[4647] https://www.libgeodecomp.org/
[4648] https://www.libgeodecomp.org/
[4649] https://www.lsu.edu
[4650] https://www.conan.io/
[4651] https://www.lsu.edu
[4652] https://uoregon.edu/

- Patrick Diehl, who worked on implementing CUDA support for our companion library targeting GPGPUs (HPXCL[4653]).
- 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)[4654]) who contributed to *HPX* the implementation of several parallel algorithms (as proposed by N4313[4655]).
- Anuj Sharma and Christopher Bross (Department of Computer Science 3 - Computer Architecture[4656]), who worked on *HPX* in the context of the Google Summer of Code[4657] program 2014.
- Martin Stumpf (Department of Computer Science 3 - Computer Architecture[4658]), who rebuilt our contiguous testing infrastructure (see the *HPX* Buildbot Website[4659]). Martin is also working on HPXCL[4660] (mainly all work related to OpenCL[4661]) and implementing an *HPX* backend for POCL[4662], a portable computing language solution based on OpenCL[4663].
- Grant Mercer (University of Nevada, Las Vegas[4664]), who helped creating many of the parallel algorithms (as proposed by N4313[4665]).
- Damond Howard (Louisiana State University (LSU)[4666]), who works on HPXCL[4667] (mainly all work related to CUDA[4668]).
- 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[4669]. He also contributed an implementation of an API similar to Fortran co-arrays on top of *HPX*.
- John Biddiscombe (Swiss National Supercomputing Centre[4670]), who helped with the BlueGene/Q port of *HPX*, implemented the parallel sort algorithm, and made several other contributions.
- Erik Schnetter (Perimeter Institute for Theoretical Physics), who greatly helped to make *HPX* more robust by submitting a large amount of problem reports, feature requests, and made several direct contributions.
- Mathias Gaunard (Metascale), who contributed several patches to reduce compile time warnings generated while compiling *HPX*.
- Andreas Buhr, who helped with improving our documentation, especially by suggesting some fixes for inconsistencies.

4653 https://github.com/STEllAR-GROUP/hpxcl/
4654 https://www.cct.lsu.edu
4655 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
4656 https://www3.cs.fau.de
4657 https://developers.google.com/open-source/soc/
4658 https://www3.cs.fau.de
4659 http://rostam.cct.lsu.edu/
4660 https://github.com/STEllAR-GROUP/hpxcl/
4661 https://www.khronos.org/opencl/
4662 https://portablecl.org/
4663 https://www.khronos.org/opencl/
4664 https://www.unlv.edu
4665 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4313.html
4666 https://www.lsu.edu
4667 https://github.com/STEllAR-GROUP/hpxcl/
4668 https://www.nvidia.com/object/cuda_home_new.html
4669 https://www.numscale.com/nt2/
4670 https://www.cscs.ch

- Patricia Grubel (New Mexico State University[4671]), 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 C++11/14.
- Anton Bikineev for contributing changes to make using `boost::lexical_cast` safer, he also contributed a thread safety fix to the iostreams module. He also contributed a complete rewrite of the serialization infrastructure replacing Boost.Serialization inside *HPX*.
- Pyry Jahkola, who contributed the Mac OS build system and build documentation on how to build *HPX* using Clang and libc++.
- Mario Mulansky, who created an *HPX* backend for his Boost.Odeint library, and who submitted several test cases allowing us to reproduce and fix problems in *HPX*.
- Rekha Raj, who contributed changes to the description of the Windows build instructions.
- Jeremy Kemp how worked on an *HPX* OpenMP backend and added regression tests.
- Alex Nagelberg for his work on implementing a C wrapper API for *HPX*.
- Chen Guo, helvihartmann, Nicholas Pezolano, and John West who added and improved examples in *HPX*.
- Joseph Kleinhenz, Markus Elfring, Kirill Kropivyansky, Alexander Neundorf, Bryant Lam, and Alex Hirsch who improved our CMake.
- Tapasweni Pathak, Praveen Velliengiri, Jean-Loup Tastet, Michael Levine, Aalekh Nigam, HadrienG2, Prayag Verma, lslada, Alex Myczko, and Avyav Kumar who improved the documentation.
- Jayesh Badwaik, J. F. Bastien, Christoph Garth, Christopher Hinz, Brandon Kohn, Mario Lang, Maikel Nadolski, pierrele, hendrx, Dekken, woodmeister123, xaguilar, Andrew Kemp, Dylan Stark, Matthew Anderson, Jeremy Wilke, Jiazheng Yuan, CyberDrudge, david8dixon, Maxwell Reeser, Raffaele Solca, Marco Ippolito, Jules Penuchot, Weile Wei, Severin Strobl, Kor de Jong, albestro, Jeff Trull, Yuri Victorovich, and Gregor Daiß who contributed to the general improvement of *HPX*.

In addition to the people who worked directly with *HPX* development we would like to acknowledge the NSF, DoE, DARPA, Center for Computation and Technology (CCT)[4672], Department of Computer Science 3 - Computer Architecture[4673], and Swiss National Supercomputing Centre[4674] who fund and support our work. We would also like to thank the following organizations for granting us allocations of their compute resources: LSU HPC, LONI, XSEDE, NERSC, and the Gauss Center for Supercomputing.

HPX is currently funded by the following grants:

- The National Science Foundation through awards 1240655 (STAR), 1339782 (STORM), and 1737785 (Phylanx). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
- The Department of Energy (DoE) through the awards DE-AC52-06NA25396 (FLeCSI) and DE-NA0003525 (Resilience). Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions

[4671] https://www.nmsu.edu
[4672] https://www.cct.lsu.edu
[4673] https://www3.cs.fau.de
[4674] https://www.cscs.ch

of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

- The Defense Technical Information Center (DTIC) under contract FA8075-14-D-0002/0007. Neither the United States Government nor any agency thereof, nor any of their employees makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.
- The Bavarian Research Foundation (Bayerische Forschungsstfitung) through the grant AZ-987-11.
- The European Commission's Horizon 2020 programme through the grant H2020-EU.1.2.2. 671603 (AllScale).

CHAPTER THREE

INDEX

- genindex